THÈSE / ENS CACHAN - BRETAGNE
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention: INFORMATIQUE
Ecole doctorale MATISSE

présentée par

Houssem Eddine Chihoub

préparée à l'unité de recherche n^o 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

Managing Consistency for Big Data Applications: Tradeoffs & Self-Adaptiveness

Thèse à soutenir à Rennes
devant le jury composé de:

Pierre Sens / rapporteur
Professeur, Université de Paris 6, France

Toni Cortes / rapporteur
Professeur des universités, Université Polytechnique de Catalunya, Espagne

Esther Pacitti / examinateur
Professeur, Université de Montpellier, France

Luc Bougé / examinateur
Professeur, ENS Cachan Antenne de Bretagne, France

Gabriel Antoniu / directeur de thèse
Directeur de recherche, INRIA Rennes, France

Maria S. Pérez / co-directeur de thèse
Professeur des universités, Université Polytechnique de Madrid , Espagne

Contents

Chapter 1

Introduction

Contents

1.1 Context

RECENTLY, data sizes have been growing exponentially within many organizations. In 2010, Eric Schmidt, the CEO of Google at the time, estimated the size of the World Wide Web at roughly 5 million terabytes of data [2] while the largest storage cluster within a corporation such as Facebook has more than 100 petabyte of Data in 2013 [122]. Data is everywhere and comes from multiple sources: social media, smart phones, sensors etc. This data tsunami, known as *Big Data*, introduces numerous complications to the different aspects of data storage and management. These complications are due to the overwhelming sizes, but also the velocity required and the complexity of data coming from different sources with different requirements at a high load variability.

In order to deal with the related challenges, many Big Data systems rely on large and novel infrastructures, as well as new platforms and programming models. In this context, the emerging paradigm of Cloud Computing offers excellent means for Big Data. Within this paradigm, users can lease on-demand computing and storage resources in a Pay-As-You-Go manner. Thereby, corporations can acquire the resources needed for their Big Data applications at a low cost when needed. Meanwhile, they avoid large investments on physical infrastructures that need huge efforts for building and maintaining them, which, in addition, requires a high level of expertise.

Within cloud storage, replication is a very important feature for Big Data. At wide area cloud scales, data is replicated across multiple data centers in order to deal with fast response and local availability requirements. Therefore, clients can request data locally from a replica within the closest datacenter and get a fast response. Moreover, geographical replication provides data durability, fault tolerance and disaster recovery by duplicating redundant data in different geographical areas. However, one issue that arises with replication is guaranteeing data consistency across replicas. In this context, insuring *strong consistency* requires huge synchronization efforts across different locations and thus, exposes the users to high network latencies. This affects the performance and the availability of cloud storage solutions. One particular alternative that has become very popular is *eventual consistency*. Eventual consistency may tolerate inconsistency at some points in time but guarantees the convergence of all replicas to the same state at a future time.

The management of consistency heavily impacts storage systems. Furthermore, with Big Data scales, the management of consistency is critical to meet performance, availability, and monetary cost requirements. Traditional storage systems and databases that implement rigorous models such as strong consistency have shown their limitations in meeting the scalability demands and the performance requirements of nowadays Big Data applications. In this context, flexible and adaptive consistency solutions that consider the application requirements and only provide the adequate guarantees, should be at the heart of the Big Data revolution.

1.2 Contributions

The contributions of this Ph.D research can be summarized as follows.

Self–Adaptive Consistency Model: Consistency when Needed, Performance when Possible

Eventual consistency is very popular within cloud storage systems as a model that provides high availability and fast responses. However, this comes at the cost of a high probability of reading stale data, as the replicas involved in the reads may not hold the most recent update. In this work, we propose a novel approach, named *Harmony*, which adaptively tunes the consistency level at runtime according to the application requirements. The key idea behind *Harmony* is an intelligent estimation model of stale reads rate. This model allows to elastically scale up or down the number of replicas involved in read operations to maintain a low (possibly zero) tolerable fraction of stale reads. As a result, *Harmony* can meet the desired consistency of the application while achieving good performance. We have implemented *Harmony* and performed extensive evaluations with the Cassandra cloud storage on Grid'5000 testbed and on Amazon EC2. The results demonstrate that *Harmony* can achieve good performance without exceeding the tolerated number of stale reads.

Cost–Efficient Consistency Management in the Cloud

Recently, many organizations have moved their data to the cloud in order to provide scalable, reliable and highly available services at a low cost. In this context, monetary cost is

extremely important factor as Cloud Computing is an economical–driven paradigm. However, it is rarely considered in the studies related to consistency management. In fact, most optimization efforts concentrate on how to provide adequate tradeoffs between consistency guarantees and performance. In this work, we argue that monetary cost should be taken into consideration when evaluating or selecting the level of consistency (number of the replicas involved in access operations) in the cloud. Accordingly, we define a new metric called consistency-cost efficiency. Based on this metric, we present a simple, yet efficient economical consistency model, called *Bismar*. *Bismar* adaptively tunes the consistency level at runtime in order to reduce the monetary cost while simultaneously maintaining a low fraction of stale reads. Experimental evaluations with the Cassandra cloud storage on the Grid'5000 testbed demonstrate the validity of the metric and the effectiveness of *Bismar* in reducing the monetary cost with only a minimal fraction of stale reads.

Consistency vs. Energy Consumption: Analysis and Investigation

Energy consumption within data centers has grown exponentially in the recent years. In the era of Big Data, storage and data–intensive applications are one of the main causes of the high power usage. However, few studies have been dedicated to the analysis of energy consumption of storage systems. Moreover, the impact of consistency management has hardly been investigated in spite of its high importance. In this work, we address this particular issue. We provide an analysis study that investigates the energy consumption of application workloads with different consistency models. Thereafter, we leverage the observations about the power and the resource usage with each consistency level in order to provide insight into energy–saving practices. In this context, we introduce adaptive reconfigurations of the storage system cluster according to the applied level of consistency. Our experimental evaluations on Cassandra deployed on Grid'5000 demonstrate the heavy impact of consistency management on the energy consumption showing a tradeoff between consistency and energy saving. Moreover, they show how reconfiguring the storage organization can lead to energy saving, enhanced performance, and stronger consistency guarantees.

Customized Consistency by means of Application Behavior Modeling

Multiple Big Data applications and services are being deployed worldwide to serve a very large number of clients nowadays. These applications differ in their performance demands and consistency requirements. Understanding such requirements at the storage system level is not possible. The high-level consistency requirements of an application are not reflected at the system level. In this context, the consequences of a stale read are not the same for all types of applications. For instance, a stale read for a Web Shop could result in serious consequences compared to a stale read within social media applications. In this work, in contrast to the related work, we focus on managing consistency at the application level rather than on the system side. In order to achieve this goal, we propose an offline modeling approach of the application access behavior based on machine learning techniques. Furthermore, we introduce an algorithm that associates a consistency policy with each application state automatically. At runtime, we introduce the *Chameleon* approach that leverages the model of the application behavior in order to provide customized consistency specific to that application. First, the application state is recognized. Then, a prediction algorithm selects the

adequate consistency policy for the expected application state at the next time period. Experimental evaluations show the high accuracy of our modeling approach, exceeding 96% of correct classification of the application states. Moreover, the conducted experiments on Grid'5000 show how *Chameleon* adapts, for every time period. According to the behavior of the application and its consistency requirements, while providing best-effort performance.

1.3 Publications

Book Chapter

- **Houssem-Eddine Chihoub**, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez, *Consistency Management in Cloud Storage Systems*, accepted **Book Chapter** to be published in the book *Advances in data processing techniques in the era of Big Data*, to be published by CRC PRESS, *end of 2013*. Editors: Sherif Sakr and Mohamed Medhat Gaber.

International Conferences

- **Houssem-Eddine Chihoub**, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez, *Harmony: Towards automated self-adaptive consistency in cloud storage*, in the proceedings of the 2012 IEEE International Conference on Cluster Computing (**CLUSTER'12**), *Beijing, September 2012*. CORE Rank A (acceptance rate 28%).

- **Houssem-Eddine Chihoub**, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez, *Consistency in the Cloud: When Money Does Matter!*, in the proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid, Computing (**CCGRID'13**), *Delft, May 2013*. CORE Rank A (acceptance rate 22%).

Posters at International Conferences

- **Houssem-Eddine Chihoub**, *Self-Adaptive Cost-Efficient Consistency Management in the Cloud*, in The 25th IEEE International Parallel and Distributed Processing Symposium (**IPDPS 2013**): **PhD Forum (2013)**, *Boston, May 2013*.

- **Houssem-Eddine Chihoub**, Gabriel Antoniu, and María S. Pérez, *Towards a scalable, fault-tolerant, self-adaptive storage for the clouds*, in the **EuroSys 2011 Doctoral Workshop**, *Salzburg, April 2011*.

Research Reports

- **Houssem-Eddine Chihoub**, María S. Pérez, Gabriel Antoniu, and Luc Bougé, *Analysis and Investigation of Consistency Management on Energy Consumption in the Cloud*.

- **Houssem-Eddine Chihoub**, Shadi Ibrahim, Yue Li, Gabriel Antoniu, and María S. Pérez, *Chameleon: Application-Specific Customized Consistency by means of Application Behavior Modeling*.

1.4 Organization of the Manuscript

The remaining part of this manuscript is organized in three parts.

The first part: presents the context of our research. Chapter 2 introduces the general background of Big Data, and presents its platforms and infrastructures. In addition, it presents the emerging paradigm of Cloud Computing as an excellent mean to deal with the complexity of Big Data management. Then, this chapter addresses the challenges of Big Data on clouds. It highlights the consistency management issue and its potential impact on the storage solutions. In this context, Chapter 3 provides a survey study of consistency management in the cloud. First, the CAP theorem and its tradeoffs are presented. Then, various consistency models are presented and discussed. Thereafter, this chapter describes some popular cloud storage systems and highlights their implementations of consistency management. Finally, the chapter introduces adaptive consistency policies as efficient models to cope with the load variability of Big Data applications in the cloud.

The second part: consists of 4 chapters that present the core contributions of this work. Chapters 4, 5, and 6 are dedicated to the consistency management and our introduced solutions on the system side. In Chapter 4, we adress the consistency–performance tradeoff. We introduce our approach, *Harmony*, to provide good performance without violating the consistency requirements of the application. While the performance level is of dramatic importance, the monetary cost of the storage service in clouds is, arguably, of equal relevance. Chapter 5 investigates the impact of consistency management on the monetary cost. Moreover, this chapter describes our approach *Bismar*. *Bismar* leverages a new consistency–cost efficiency metric in order to provide a consistency management that reduces the monetary cost and provide adequate consistency guaratntees in the cloud. Similar to Chapters 4 and 5, Chapter 6 explores the impact of consistency models on the energy consumption of the storage system cluster. Moreover, in this chapter, we show the energy savings of adaptive reconfigurations (according to every consistency level) of the storage system cluster. In order to complement our work on the system side, Chapter 7 introduces *Chameleon*, our consistency management approach at the application level. We propose a behavior modeling approach of Big Data applications. The model, in addition to the application semantics, is leveraged to provide customized consistency that selects the adequate consistency policy at each time period according to the application requirements.

The third part: consists of Chapter 8. In this chapter, we summarize our contributions and present our conclusions about the management of consistency for Big Data applications in the cloud. Moreover, we discuss the limitations in this context and describe the perspectives in this area that can lead to even more efficient storage management in the cloud.

Part I

Context: Consistency Management for Big Data

Chapter 2

Big Data Systems and Cloud Computing: A Short Overview

Contents

In the recent years, data sizes have been growing exponentially within many corporations and organizations [122, 136]. The total data amount produced in 2010 is estimated to be over one zettabyte (10^{21} bytes) while it is predicted to grow by a factor of 50x over the next decade [136]. This Data Deluge phenomenon, later known as Big Data, introduces many challenges and problems. Therefore, experts deal with issues that relate to how to store, manage, process and query data on a daily basis. These problems were dealt with for many years at big corporations such as Google, Amazon, and Microsoft, relying on innovative software platforms but mainly large-scale infrastructures. On the other hand, small companies remained unequipped. The continuous growth rate of data size and variety demonstrates the

necessity for innovative and efficient ways in order to address the emerging problems. Organizations should expect to handle and process overwhelming amounts of data that exceed their capacities. In this context, Cloud Computing offers excellent tools to deal with the most challenging aspects of Big Data. In this chapter, we introduce the Big Data phenomenon as well as platforms and infrastructures to deal with the related challenges. Then, we zoom on Cloud Computing as an efficient infrastructure for leveraging Big Data management. We finally, discuss Big Data issues and challenges and how major cloud vendors tend to deal with them. Thereby, we highlight replication in the context of Cloud Computing, its features and issues.

2.1 Big Data

2.1.1 Big Data Definitions

Big Data is more than growing sizes of datasets, but rather the complexity that such a growth generates with regard to different aspects of data handling. In [133], Stonebraker introduces Big Data according to the "3Vs" model [1]. Big Data refers to datasets that are of a big volume, need big velocity, or exhibit big variety.

Big Volume. Nowadays, data sizes are exponentially increasing. Multiple companies and organizations are experiencing Data Deluge phenomenon due to multiple factors such as stored data collected over the years, data streaming and sharing over social media, increasing sensors collected data, etc. [31]. Stonebraker considers the Big Volume property as important and challenging for two types of analytics: "small analytics" and "big analytics". Small analytics include smaller operations such as running SQL analytics (*count*, *sum*, *max*, *min*, and *avg*), while big analytics are more complex operations that can be very expensive on very large datasets, such as clustering, regressions, and machine learning.

Big Velocity. For many organizations, the most challenging aspect of Big Data is not exclusively the large volume. It is rather, how fast to process data to meet demands [1, 31]. A wide range of applications require fast data processing in near real-time manner, no matter how overwhelming the size of data. For instance, electronic trading, RFID tags and smart metering, ad placement on Web pages, fit in this class of applications.

Big Variety. Data comes in various formats: from text data to video and audio, from traditional DBMS (DataBase Management Systems) formats to semi-structured data (*e.g.* XML), to large binary objects. The primary challenge is to integrate all these types of data, and manage them in an efficient way. SAS [31] estimates that 80% of organization data are not numerical. Nevertheless, those data should be included in analytics and decision–making.

In addition to the aforementioned 3 dimensions of Big Data in the form of volume, velocity and variability, SAS [31] defines two complementary dimensions.

Big Variability. In addition to velocity and variety, data may exhibit Big Variability in the form of inconsistent or non-regular data flows with periodic peaks. In particular, this

is a typical behavior for applications such as social media that might suddenly experience high loads during trending news or with seasonal events. In this context, Big Data variability might be particularly challenging to manage, for instance when social media is involved.

Big Complexity. Managing huge volumes of data that come from multiples sources and systems is most of the times difficult. Management aspects can be particularly challenging for linking, matching, cleansing and transforming data across systems and platforms.

2.1.2 Big Data Platforms

Traditional storage systems such as DBMS and data querying models fail to meet Big Data requirements. In order to address this issue, multiple storage solutions and data processing models were introduced.

2.1.2.1 Parallel File Systems

Parallel file systems were introduced in order to overcome centralized file systems scalability and failure tolerance limitations. They rely on decentralized storage that enables scalable performance and fast access.

The massively parallel design of this class of file systems allows the distribution of workloads over multiple servers, that might be spread over wide areas, for data accesses and sometimes metadata accesses, thus providing scalable performance. Moreover, thanks to replication, the file systems might have faster access when reading data from closer replicas.

In order to provide such scalability features, most parallel file system designers are reluctant to stick to POSIX semantics. POSIX standard, much like ACID requirements, imposes a strong semantics and makes various constraints on data handling that are penalizing for system performance and present a primary bottleneck for the system scalability. However, many file systems such as GPFS [113] consider that POSIX compliance is very important in spite of the performance overhead. Other systems tend to provide a minimal POSIX-compliant data access interface even though the system itself is not fully POSIX-compliant, like Luster file system [114]. On the other hand, modern file systems, such as PVFS [38], Ceph [129], Google File System [63], and HDFS [69], trade POSIX compliance for better performance, scalability and availability.

2.1.2.2 NoSQL Datastores

Over the last few years, RDBMS (Relational DataBase Management Systems) have shown their scalability limits to face Big Data overwhelming. RDBMS designs and query models (mainly SQL) are based on strong ACID semantics (Atomicity, Consistency, Isolation, Durability) that for many years were considered as unavoidable in order to provide a "correct" model. However, with the growing scales of applications, in the era of Big Data, imposing such semantics presents unprecedented limitations [35]. The NoSQL movement rejects part or all of the ACID semantics in order to provide scalability, availability and high performance. NoSQL datastores generally rely on a much simpler data queering models than SQL, where accesses are key based and data are schematized in a key/value or key/values

manner. This enables data to be stored and served in a massively distributed way based on adequate distribution algorithms and data structures. Nowadays, NoSQL data stores and storage systems such as Amazon Dynamo [49], Google Big Table [40], and Cassandra [85, 18] are proven to be very efficient over scaling to serve data at the planet scale.

These systems are usually designed for a specific class of applications (*e.g.* Web applications, document storage, etc.). Accordingly, they rely on designs that might relax or forfeit part of the strong semantics in order to trade it for better performance, availability, and scalability. For instance, Amazon Dynamo relaxes strong consistency for several services that do not strictly require it, and trades it for very high availability and fast data access. Moreover, these systems are, in general, easily integrated with Hadoop for more complex queries and analytics at a wide scale in real-time.

2.1.2.3 MapReduce

MapReduce [48] is a programming model introduced by Google to provide efficient processing for large data sets in a massively parallel way. MapReduce frameworks consist of two main phases: the *Map* phase and the *Reduce* phase. The users specify a *map* function that divides the input into sub-problems and generates intermediate data in the form of key/value pairs. These intermediate data are further passed to a reduce function also specified by the users. The *reduce* phase merges all the values associated with the same intermediate key.

Many implementations have appeared since the introduction of MapReduce and its implementation within Google. Google implementation can be used within Google App Engine cloud [67]. The most popular implementation of MapReduce is Hadoop [20]. Hadoop gained major support from multiple corporations and organizations since its introduction, which promoted MapReduce as an efficient model for Big Data processing. The Hadoop project is an open-source collection of Apache sub-projects. They include Apache Hadoop MapReduce framework, Apache HDFS file system [69], and HBase [21] datastore among others. Hadoop is widely used nowadays at companies that deal with data deluge. Yahoo! was one of the first corporations to sponsor Hadoop and use it in production to produce data for Yahoo! Web search [70]. In 2011, Facebook claims that their Hadoop deployment with *Corona* [122] is the largest one in production, roughly processing 30 PB of data (100 PB in 2013). Moreover, cloud vendors provide their clients with Hadoop-based services that are very simple to deploy and to put into production such as Amazon Elastic MapReduce [10].

MapReduce has become the natural data processing model for Big Data as it provides scalability, availability, performance, and reliability whereas traditional RDBMS (with their SQL-based query model) fail in meeting the scalability demands. MapReduce frameworks such as Hadoop are able to process data efficiently at large scales to replace SQL complex queries most of the time. In this context, a wide range of applications were developed within a MapReduce framework including data distributed querying applications, data analytics applications, parallel sort applications, and data clustering and machine learning applications.

2.1.3 Big Data Infrastructures

The phenomenal growth of Big Data requires efficient infrastructures and novel computing paradigms in order to meet the huge volumes of data and their big velocity. Hereafter, we

present three common infrastructure models that offer excellent means to manage Big Data.

2.1.3.1 Clusters

Cluster computing consists in connecting a set of computers with their regular software stack (*e.g.* operating systems) together through, commonly, local area networks (*LAN*) to build one global system. Cluster computing was one of the primary inspirations for efficient distributed computing. The primary aim of this computing paradigm is to provide better performance, and availability at a low cost.

Over the years, cluster computing tools and paradigms evolved to reach high efficiency. Nowadays, building a cluster has become easy with support for efficient job scheduling, and load balancing. Moreover, cluster users can rely on a mature stack of standardized and dedicated tools such as Linux operating systems, message passing interface (MPI) tools [98], and various monitoring tools to run a wide range of applications.

In the era of Big Data, it is common for organizations to build dedicated clusters that usually consist of commodity hardware. These clusters run Big Data platforms (such as Hadoop and NoSQL systems), to host dramatically large volumes of data efficiently. They enable meanwhile, real-time processing and high degree of availability. Such cluster sizes may vary from tens of nodes to tens of thousands of nodes [122].

In the area of high performance computing (HPC), Supercomputers that consist of hundreds of thousands of cores can run high-performance applications that exhibit high I/O demands. These supercomputers have dedicated massively-parallel architectures to meet the high computation demands and the increasing storage needs of HPC.

2.1.3.2 Grids

In their attempt to define "Grid Computing", Foster and Kesselman qualify a computational grid as "*a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.*" [61]

Four years later, Ian Foster reformulates his definition as a checklist [60]. Accordingly, a grid is a system that:

- Provides coordination over resources for users with no centralized control.
- Provides standard, open general-purpose protocols and interfaces.
- Deliver nontrivial qualities of service.

Grid Computing addresses, generally speaking, the federation and the coordination of heterogeneous sub-organizations and resources that might be dispersed over geographically remote areas. It aims at hiding the complexity of the coordination of sub-systems from the users and exposes them the illusion of one global system instead.

2.1.3.3 Clouds

Parallel and distributed computing have been evolving over the years so as to provide better quality of service to users while insuring efficiency in terms of performance, cost, and failure tolerance. A newly emerged computing paradigm is Cloud Computing. In this paradigm, software and/or hardware are delivered as a service over a computer network, typically internet. Cloud services differ in their abstraction level. They can be categorized into three levels: Infrastructure, Platform, and Software.

Since its emergence, Cloud Computing has immediately been adopted by Big Data applications. For many organizations, acquiring resources in a Pay-as-You-Go manner in order to scale out their Big Data platforms when required, was long-time needed, and the cloud providers offer just that. Moreover, most of cloud providers offer Big Data platforms as a service. They enable clients to run their applications without worrying about the infrastructure management and its costly maintenance. Nowadays, it is just natural for emerging companies, such as *Netflix* [27] that offers a video on-demand service, host and manage all their data in the cloud (Amazon EC2 [9]).

In the next section, we further highlight this paradigm of Cloud Computing as a natural and excellent mean for Big Data.

2.2 Cloud Computing

Cloud Computing is the on-demand delivery of IT resources via the Internet with Pay-As-You-Go pricing [134]. As a newly emerged computing paradigm, several attempts to define Cloud Computing were introduced [25, 130, 36]. *Buyya et al.* define a cloud as "*a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers.*"

In this context, Cloud Computing is an economy -and business- driven paradigm. The cloud provider and the customers establish an SLA (Service Level Agreement) contract that formally defines the service delivered. The customers can specify their needs in terms of resources, performance, availability, etc., and pay only for what they consume.

In addition to its flexible and fair pricing models, the Cloud Paradigm promises highly desired features and advantages. The main two features are *on-demand resources delivery* and *elasticity*. Customers can easily purchase new resources from the cloud provider in real time and scale up horizontally, and simply free leased resources later according to their requirements and peak computation times. Furthermore, leasing resources from cloud providers with high–level expertise on how to manage infrastructures and platforms, guarantees high rates of reliability and availability. Cloud Computing presents an excellent model of resource sharing as well. The computing resources delivered over Internet are, usually, running on virtual machines running side by side on the same physical infrastructure. In this context, advances in networking and virtualization technologies enabled applications and services to share infrastructure resources efficiently.

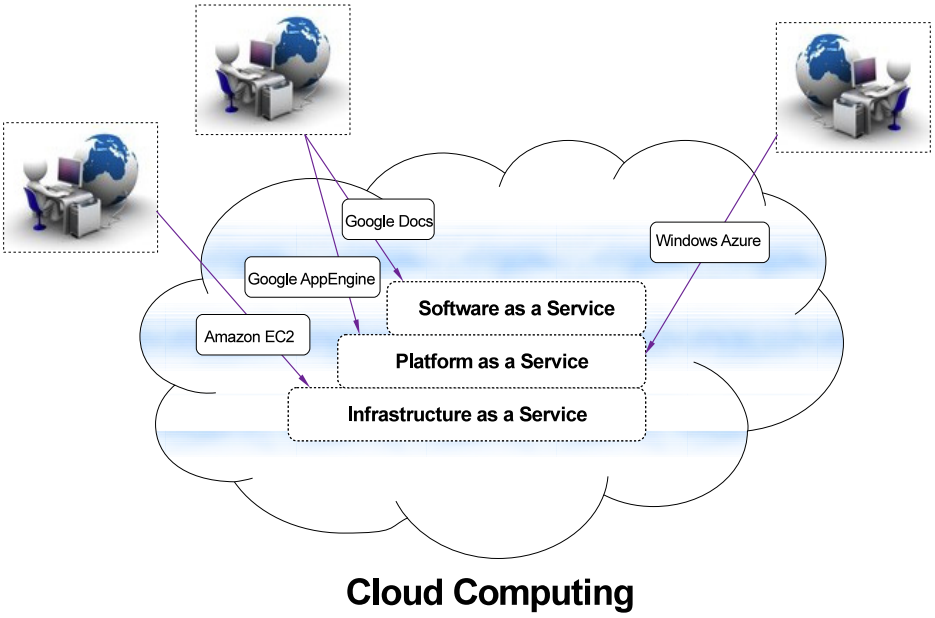


Figure 2.1: Abstraction layers of cloud services

2.2.1 Cloud Service Levels

Cloud Computing services differ in their levels and can be classified into three layers as shown on Figure 2.1. Cloud clients nowadays can purchase resources of different natures from various providers. A client can lease infrastructure resources, platform resources, or software resources, possibly all three types simultaneously.

Infrastructure as a Service (IaaS)

In this type of service, clients outsource the infrastructure and the equipment necessary for their platform and application deployments. Clients can just rent on-demand the virtual machines (VMs), and the storage resources they need. The cloud provider hosts, manages, and maintains pools of infrastructure equipment including storage, computing servers, networking, and other hardware components. The task of managing and maintaining the software stack is left to the clients. The typical pricing models for such services are Pay-As-You-Go based. Examples of this type of service include Amazon Elastic Cloud Compute (EC2) [9] and HP Cloud [74].

Platform as a Service (PaaS)

This type of service includes providing a computing platform, in addition to the implicit infrastructure that hosts the platform resources. Users rely on the provided platform to create and/or run their applications. PaaS has been gaining a major popularity in recent years. Clients export the platform and infrastructures management, with all its complexity, to a third party that is, generally, better experienced, and better equipped to deal with the

related platform and infrastructure challenges. For instance, any client can deploy Hadoop MapReduce applications in a few mouse clicks over Amazon Elastic MapReduce service [10].

Software as a Service (SaaS)

Software as a service is the highest level in the abstraction of service layers. The software services are hosted on the infrastructure of the cloud provider in this model, and are delivered over a network, Internet typically. With the evolution of the network hardware and the high bandwidth recently achieved, it is possible for clients to run applications and software over the Internet that were traditionally run in local computers. Such a type of service is beneficial to client organizations as it facilitates the administration of applications, allows better homogeneity of versions between application users, and more importantly provides global accesses and collaborations.

2.2.2 Cloud Computing Models

Cloud Computing platforms vary in their targeted customers. Therefore, multiple cloud models were introduced to fit customer ranges.

Public Clouds

Public cloud providers offer the access to services to a wide public over Internet. Therefore, many individuals and organizations can benefit from the provided services and share physical resources. This model is very popular, with major Cloud providers such as Amazon, Microsoft, and Google offering reliable and efficient services to a wide range of clients at arguably low costs. The public cloud paradigm provides many advantages over other paradigms. In particular, public clouds significantly reduce the financial cost, especially for new projects, as the bill exclusively includes the amount of resources usage. Moreover, in this model, the client is offered high on-demand elasticity and scalability, as cloud providers create the illusion of infinite resources.

Private Clouds

Private clouds are platforms specifically built to serve one organization. The cloud platform might be built internally by the same organization or by a third party. Private cloud flexibility may be beneficial to organizations as the platform can be built to fit the specific requirements of that organization. However, the organization is obliged to handle (by itself or by delegating to a third party) the infrastructure and the platform management and maintenance, which can be both complicated and costly. Moreover, private clouds might suffer from limited scalability and elasticity as the needs grow over the years. One particular advantage of private clouds is their potential high security and privacy as only customers belonging to the organization are allowed to access the cloud.

Hybrid Clouds

Hybrid clouds are the combination and federation of two or more clouds of different nature (public and private). IT organizations might require extra resources at peak times. One solution is to acquire additional resources from public cloud providers. Such phenomenon is called "Cloud Bursting" and enables cross–clouds scalability. Hybrid clouds offer some degree of scalability and elasticity. They help organizations to keep critical data and workloads in-house. In contrast, the heterogeneity of resources belonging to multiple clouds may introduce an additional management overhead and performance variability. Although, virtualization techniques achieved sufficient maturity to deal with the heterogeneity of physical resources. Moreover, the combination of private and public resources may, in some cases, present a security vulnerability to the private data.

2.2.3 Cloud Computing Platforms

Since the emergence of the Cloud Computing paradigm, many corporations and research organizations invested resources in building efficient cloud platforms. In this section, we review a few major and emerging cloud platforms both in industry and academia.

2.2.3.1 Public Cloud Services

Amazon Web Services. Amazon Web Services (AWS) [14] are the Cloud Computing services offered by Amazon for their public clients over Internet. The collection of Amazon Web Services include computing, storage and network services at multiple levels from infrastructures to platforms. Traditionally, AWS are of IaaS type. However, they evolved over the years to include PaaS services as well. The main computing service is Amazon Elastic Cloud Compute (EC2) [9]. EC2 provides virtual machines called *instances* that may come with a preferred operating system. Clients can use these instances to deploy a wide variety of applications with an entire control over the software stack. In addition, AWS provides multiple storage services. The service to store and manage unstructured data is Amazon Simple Storage Service (S3) [13]. S3 allows users to store their data *objects* in the cloud in *buckets*. In contrast, users that want to attach virtual storage devices to their virtual machines in EC2 can rely on Elastic Block Store (EBS) volumes [8]. In order to support structured data storage, AWS introduced the Amazon DynamoDB [7] service. DynamoDB is a NoSQL storage service that relies on the Amazon Dynamo [49] system backend and serves clients over the Internet via a RESTful API.

Microsoft Windows Azure. Windows Azure [95] is a Microsoft cloud service platform. Windows Azure provides a wide range of PaaS services for its clients both at the infrastructure level and the platform level. These services include platforms for creating websites, and platforms for deploying and managing large-scale and scalable applications. Moreover, within Azure, it is now also possible to lease resources of infrastructure nature (IaaS) over Internet. Virtual machines that run either a Windows or Linux can be leased. Data management services within Azure include a variety of features supporting SQL data management with Windows SQL server databases and Windows Azure storage for large binary objects (BLOBs).

Google App Engine. App Engine [67] is Google's Platform as a Service cloud. It allows customers to build their web applications in the cloud and run on the same infrastructure that powers Google applications. App Engine provides tools that automatically and elastically scale up resource provisioning for deployed applications. The supported programming languages of the platform are Python, Java, and Go [120]. Data storage and management is orchestrated by the *High Replication Datastore (HRD)*. HRD supports a hierarchically structured space where objects are called *entities* and each *entity* might have a parent *entity*

2.2.3.2 Open-Source Toolkits for Managing Cloud Infrastructures

Since the emergence of the Cloud Computing paradigm, various toolkits to manage the cloud infrastructures were introduced. Tools such as Eucalyptus [57] for building EC2-compatible private and hybrid clouds, and Nimbus [99] for building scientific clouds gained major success. In this section, we present two illustrative cloud toolkits: OpenStack that has become very popular serving in production within many organizations, and OpenNebula the European cloud toolkit that targets the standardization of cloud querying models.

OpenStack. OpenStack [103] is an open-source Infrastructure as a Service platform for public and private clouds under Apache 2.0 license. OpenStack aims at providing a massively scalable and efficient open source cloud solution for corporations, service providers, small and medium enterprises, and researchers. The OpenStack architecture consists of three major components. *Swift* is a store that supports basic objects storage. Cloud environments in production that are based on OpenStack most often build file system solutions based on this component. *Glance* is a component, which provides the storage and management support for virtual machine images. The computing service is provided by the *Nova* component. Similar to Amazon EC2, Nova provides virtual machines (VMs) on-demand supporting full elasticity and scalability. OpenStack is being widely used in production within many organizations. Companies like *PayPal* and *Intel*, or institutions such as Argonne National Laboratory, US Department of Energy, and *San Diego Supercomputer Center* are known users of OpenStack.

OpenNebula. OpenNebula [102] is an open-source cloud computing toolkit. OpenNebula toolkit orchestrates easy deployments of Infrastructure as a Service nature to form several types of clouds including private, public, and hybrid clouds. The toolkit fully supports efficient management of virtualization over computing, storage, and network resources. The aim of OpenNebula design is to provide a standardization of IaaS clouds with a full support of interoperability and portability. OpenNebula provides two main cloud interfaces. The EC2 interface and the OGF OCCI [101] interface. The latter was first proposed as an attempt toward a standardization of the cloud query model. OpenNebula clients can use either interfaces to acquire virtual machines over the network. Moreover, the OCCI interface supports data access over a file system. In this context, OpenNebula providers can easily integrate their preferred file system or storage support.

2.3 Big Data Applications in the Cloud: Challenges and Issues

The alarming growth rate of Big Data introduces unprecedented challenges and problems to IT experts. One of the first and major problems was of infrastructure nature. Companies and organizations face an overwhelming data tsunami that exceeds their infrastructure capacities. The emergence of the Cloud Computing paradigm provided a safety net as it provides an infinite amount of resources on-demand over Internet at a fair price. Moreover, major cloud vendors that possess the right tools and experience to deal with Big Data, provide storage solutions and Big Data platforms to their clients. Henceforth, cloud clients delegate a major part of Big Data management complexity to the cloud providers. This significantly reduces the cost as the management of Big Data at home requires large investments in infrastructures, platforms and human resources that many organizations cannot afford. In this context, Amazon Web Services (AWS) for instance provide their clients with mature infrastructures and tools to deal with Big Data from day one. It is fairly easy nowadays, to store one's data on Amazon S3 or Amazon DynamoDB. Moreover, Amazon provides Elastic MapReduce service that allows clients to run Hadoop applications on their data without worrying about the deployment nor the management of the platform.

2.3.1 Big Data Challenges

Big Data challenges are inherited from problems that storage and database experts have been facing for long years. They learnt how to deal with them at smaller scales. However, early methods fail, in most, to meet nowadays enormous scales. In the following, we present main challenges of the Big Data community and how organizations tend to deal with them in the cloud.

Data Durability

Data durability is a primary concern for Big Data applications. Data loss may have various consequences from disasters with critical applications, to merely small misfunctions. Over the years, storage architects proposed innovative ways to insure data durability relying on a wide range of techniques that include replication and log-based mechanisms. One approach that is very popular at the level of disk storage is RAID (Redundant Array of Independent Disks) [105]. RAID techniques combine multiple disk drives with multiple data distribution algorithms to ensure data durability. However, such a technology fails to meet the level of actual cloud scales [108]. Intel for instance, proposes to replace these techniques with erasure code mechanisms [108]. Cloud storage systems such as Yahoo! PNUTS [44] and Amazon Dynamo [49] on the other hand rely (at a higher level) on *hinted writes* (based on logs) and geographical replication to ensure data durability. Glacier [11] is an Amazon service to provide long-term storage. Amazon Glacier promises 99.999999999 th percentile of data durability. Its architecture relies on a huge amount of tape libraries that are geographically replicated. However, Amazon Glacier is not an online system as retrieving data requires hours.

Fast Access

The growth of data in volume, variety, and velocity is associated with a growing need for high performance and fast access. To address this challenge, given the enormous volumes of data, various techniques are put to use by cloud providers and Big Data corporations. Technological advances in Flash Memory and Solid-State Drives (SSD) reduce the access time by about 8 times [71] compared to classical hard drives. At the software level, novel storage system architectures rely on replication in order to provide faster access to local copies of data. In particular, Optimistic Replication [110] provides much faster access in the case of geographical distribution as data can be accessed directly from geographical close replica while asynchronously propagated to other replicas. Another technique that enables faster access is enabling in-memory computation within distributed storage systems. This enables storage users to access most of their data in memory (RAM) instead of disks. Systems such as Google BigTable [40] and Cassandra [85] rely on in-memory data structures called *memtables*, that provide fast access directly from memory for a large portion of data.

Data Availability

Data is said to be available if all (or most) access requests result in a successful response [66, 35]. Availability is a critical requirement for many Big Data applications. Web services such as Web shops require a high degree of availability as customers must always get a response for their requests to avoid financial losses. For instance, the cost of a single hour of downtime for a system handling credit card sale authorizations has been estimated to be between $ 2.2 M–$ 3.1 M [106]. In this context, novel architectures that guarantee high levels of availability at a wide scale were introduced. The Amazon.com e-commerce platform [15], along with many of Amazon Web Services, relies on Amazon Dynamo storage [49] to achieve extremely high levels of availability. Amazon Dynamo forfeits few of the ACID strong semantics in order to offer a Service Level Agreement (*SLA*) guaranteeing that a response will be provided within 300 ms for 99 % for a peak client load of 500 requests per second [49].

Failure Tolerance & Disaster Recovery

Cloud storage infrastructures, commonly, consist of commodity hardware. Therefore, components failures are the norm rather than the exception. One of the main challenges for Big Data platforms and infrastructures is to keep operational in the presence of one or more failing components. Redundancy and replication are known ways to handle fault tolerance. Solutions such as RAID (Redundant Array of Independent Disks) are widely used. However, the scalability of these techniques is a major issue. One popular solution that scales to wide areas level is Optimistic asynchronous Replication [110]. This technique is an alternative to synchronous replication, that enables scalable performance while providing fault tolerance. Furthermore, replicating data at multiple geographical areas and insuring its durability provides disaster recovery capacity. Nowadays, it is fairly easy and cheap to replicate data across multiple continents within major cloud providers. In this context, systems are able to recover even with the compromise of most or all the data centers in a geographical area subject to a sinister.

2.3.2 Our Focus: Replication and Consistency

Replication. Replication is a very important feature for Big Data. It consists in storing multiple copies of the same data on multiple storage devices or storage nodes. In the era of planet-scale services and applications, replication is mandatory to address the Big Data challenges. It provides the necessary means to insure data durability, to enable a fast access through local replicas, to increase the level of data availability, and to guarantee fault tolerance. Traditionally, data propagation from one replica to the others is performed in a synchronous way. Fundamentally, for each update or write, data are immediately propagated to other replicas before returning operation success. However, this approach showed its limitation with today's growing scales, as it fails to provide scalable performance, particularly at wide scale replication. The need for synchronization coupled with high network latencies badly impacts the operation latencies as reads and writes must await responses from all replicas no matter their locations before returning a success. Optimistic (or lazy) Replication [110, 84, 83] was proposed to overcome this particular problem. Data propagation to other replicas (or a subset of replicas) is performed lazily and delayed to a future time. In this case, update and write operations are allowed to return a success after data are committed to the local replica whereas they are yet to be fully propagated. This situation may lead to a temporary replica divergence. Meanwhile, Optimistic Replication allows better performance. The latency of read and write operations is not affected by the network latency. Moreover, the throughput of the storage system is not slowed down by the extra traffic generated by an immediate data propagation to other replicas. However, this approach presents a major, complex issue: that of *data consistency*.

Consistency. Insuring data consistency across replicas is a major issue. Traditional databases (RDBMS) and file systems are bounded by rigorous semantics (ACID for RDBMS and POSIX for filesystems) where a strong data consistency is one of the primary requirements. Strong consistency is, usually, achieved relying on synchronous replication and guarantees that replicas of the same data are always perceived in the same state by the users. Within early systems scales, this was not an issue. However, within Big Data systems and applications, strong consistency may be too costly introducing a heavy performance overhead and reducing data availability. To overcome such a limitation, eventual consistency [124, 115, 119] has become very popular in the recent years. It may allow some temporal inconsistency at some points in time, but insures that all replicas will converge to a consistent state in the future. This is acceptable for many modern Big Data applications, in particular a wide range of Web applications for which update conflicts are not common and easily solvable. Moreover, by relying on Optimistic Replication, eventual consistency provides better performance and higher availability. In this context, various tradeoffs regarding consistency emerge. They include the Consistency-Perfroamnce tradeoff and the Consistency-Availability tradeoff among others. As a result, consistency management is crucial within storage systems and their capacity to deal with Big Data challenges. In the next chapter, we survey consistency management in the cloud. We present consistency tradeoffs, consistency models, their implementation in modern cloud storage, and adaptive consistency management approaches.

2.4 Summary

The explosion of data volumes in recent years has revolutionized the way IT organizations manage their hardware and software resources. Henceforth, novel computing paradigms and data processing platforms were introduced. To cope with the challenges of the emerging Big Data phenomenon, a few infrastructure models such as Cluster, Grid, and Cloud Computing provide the necessary means to efficiently host, transfer and process data. In particular, the Cloud Computing paradigm offers the on-demand flexibility and elasticity of resource leasing that Big Data platforms critically need. It introduces various models and service levels to cope with the variety of Big Data applications. Moreover, in order to deal with data availability, data durability, fast access, and fault tolerance challenges, replication is an important feature in the cloud. However, replication arises the issue of data consistency across replicas. In this context, we highlighted the consistency management issue, and its heavy impact and consequences on dealing with Big Data challenges. In the next chapter, we discuss in further details the importance of consistency management, the related trade-offs and the state of the art of consistency model implementations within modern storage systems.

Chapter 3

Consistency Management in the Cloud

Contents

This chapter is mainly extracted from the book chapter: *Consistency Management in Cloud Storage Systems* Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, Maria S. Pérez. In the book *Advances in data processing techniques in the era of Big Data*, to be published by CRC PRESS, end of 2013. Editors: Sherif Sakr and Mohamed Medhat Gaber.

REPLICATION is a key feature for Big Data applications in the cloud in order to cope with Big Data related challenges. In this context, replication is used for fast accesses, high levels of availability, data durability, fault tolerance, and disaster recovery. However, as highlighted in the previous chapter, replication raises the issue of data consistency across replicas. Hereafter, in this chapter, we address this issue of consistency management in cloud storage systems. We survey consistency models and their impacts on storage system features in the context of Cloud Computing. First, we highlight the different tradeoffs related to consistency as introduced by the *CAP* theorem and beyond. Then, we present various consistency models introduced over the years with their provided guarantees. Thereafter, we discuss the implementation of consistency mechanisms within some of the most popular cloud storage systems. In order to conclude this chapter, we present academic approaches that tend to manage consistency dynamically at application runtime in order to guarantee consistency when needed and enhance performance when possible.

3.1 The *CAP* theorem

In his keynote speech [35], *Brewer* introduced what is known as *the CAP* theorem. This theorem states that at most only two out of the three following properties can be achieved simultaneously within a distributed system: *Consistency*, *Availability* and *Partition Tolerance*. The theorem was later proven by *Gilbert and Lynch* [66]. The three properties are important for most distributed applications such as web applications. However, within the *CAP* theorem, one property needs to be forfeited, thus introducing several tradeoffs. In order to better understand these tradeoffs, we will first highlight the three properties and their importance in distributed systems.

Consistency: The consistency property guarantees that an operation or a transaction is performed atomically and leaves the systems in a consistent state, or fails instead. This is equivalent to guaranteeing both the atomicity of operations and the consistency of data properties (*AC*) of the *ACID* (Atomicity, Consistency, Isolation and Durability) semantics in relational database management systems (*RDBMs*).

Availability: In their *CAP* theorem proof [66], the authors define a distributed storage system as continuously available if every request received by a non-failing node must result in a response. On the other hand, when introducing the original *CAP* theorem, *Brewer* qualified a system to be available if *almost* all requests receive a response [35]. This is the case of nowadays SLAs (Service Level Agreement) proposed by Cloud Computing vendors insuring availability rates that exceeds 99.99% of operations.

Partition Tolerance: In a system that is partition tolerant, the network is allowed to loose messages between nodes from different components (data centers for instance). When a network partition appears, the network communication between two components (racks, data centers etc.) is off and all the messages are lost. Since replicas may be spread over different partitions in such a case, this property has a direct impact on both consistency and availability.

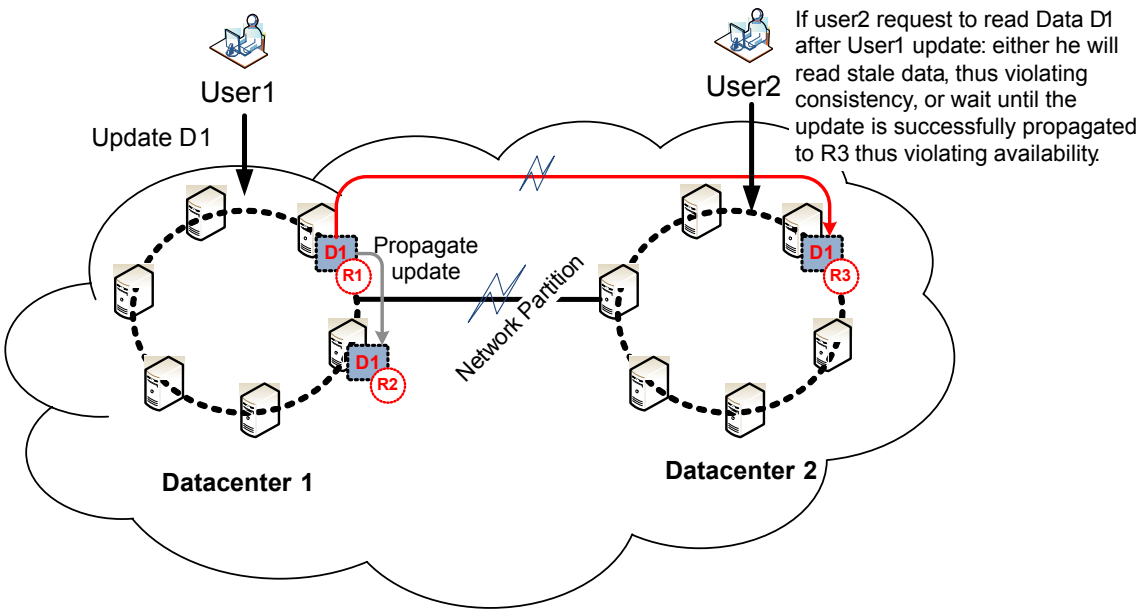


Figure 3.1: **Consistency *vs.* Availability in Geo-replicated Systems**

The implications of the *CAP* theorem introduced challenging and fundamental tradeoffs for distributed systems and service designers. Systems that are designed to be deployed on single entities, such as an RDBM, aim to provide both availability and consistency properties since partitions are not an issue. However for distributed systems that rely on networking, such as geo-replicated systems, partition tolerance is a must for a big majority of them. This in turn introduces, among other tradeoffs derived from the *CAP* theorem, the *Consistency vs. Availability* as a major tradeoff. As shown in Figure 3.1, user requests can be served from different replicas in the system. If partitions occur, an update on one replica cannot be propagated to other replicas on different partitions. Therefore, those replicas could be made either *available* to the clients, thus violating consistency, or otherwise, made *unavailable* until they converge to a *consistent* state, which can happen after recovering from the network partition.

Beyond the *CAP* Theorem

The proposal of the *CAP* theorem a few years ago had a huge impact on the design of distributed systems and services. Moreover, the ever-growing volume of data along with the huge expansion of distributed systems scales makes the implications of the *CAP* theorem of even higher importance.

Twelve years after the introduction of his *CAP* theorem, *Brewer* still ponders its implications [34]. He estimates that the theorem achieved its purpose in the past in the way it brought the community's attention to the related design challenges. On the other hand, he judges some interpretations of the implications as misleading, in particular, the two–out–of–three tradeoff property. The general belief is that the partition tolerance property P is insurmountable for wide-area systems. This often leads designers to completely forfeit consistency C or availability A for each other. Given that partitions are rare. *Brewer* states that

the modern goal of the *CAP* theorem should be to maximize combinations of *C* and *A*. In addition, system designers should develop mechanisms that detect the start of partitions, enter an explicit partition mode with potential limitations of some operations, and finally initiate partition recovery when communication is restored.

Abadi [3] states as well that the *CAP* theorem was misunderstood. *CAP* tradeoffs should be considered under network failures. In particular, the Consistency-Availability tradeoff in *CAP* is for when partitions appear. The theorem property *P* implies that a system is partition-tolerant and more importantly, is enduring a partition. Therefore, and since partitions are rare, designers should consider other tradeoffs that are, arguably, more important. A tradeoff that is more influential, is the latency-consistency tradeoff. Insuring strong consistency in distributed systems requires a synchronized replication process where replicas belong to remote nodes that communicate through a network connection. Subsequently, reads and updates may be costly in terms of latency. This tradeoff is *CAP-Independent* and exists permanently. Moreover, *Abadi* makes a connection between latency (response time of an operation) and availability. When latency is higher than a specific timeout the system becomes unavailable. Similarly, the system is available if the latency is smaller than this timeout. However, the system can be available and exhibit high latency nonetheless. For these reasons, system designers should consider this additional tradeoff along with *CAP*. *Abadi* proposes to unify the two in a unified formulation called *PACELC* where *PAC* refers to the *A* (availability) and *C* (consistency) tradeoff *if* a partition *P* exists, and *ELC* refers to *else*: (*E*), in the absence of partitions, the latency *L* and consistency *C* tradeoff should be considered.

After they proved the *CAP* theorem, *Gilbert and Lynch* reexamined the theorem properties and its implications [65]. The tradeoff within *CAP* is another example of the more general tradeoff between *safety* and *liveness* in unreliable systems. Consistency can be seen as a *safety* property for which every response to client requests is correct. In contrast, availability is a *liveness* property that implies that every client request would eventually receive a response. Hence, viewing *CAP* in the broader context of safety-liveness tradeoffs provides insight into the feasible design space for distributed systems [65]. Therefore, they reformulate the *CAP* theorem as follows: "*CAP states that any protocol implementing an atomic read/write register cannot guarantee both safety and liveness in a system prone to partitions*". As a result, the practical implications dictate that designers opt for best-effort availability, thus guaranteeing consistency, and best-effort consistency for systems that must guarantee availability. A pragmatic way to handle the tradeoff is by balancing the consistency-availability tradeoff in an adaptive manner. We will further explore this idea in Section 3.4.

3.2 Consistency Models

In this section, we present multiple consistency models. We particularly, focus on *strong consistency* and *eventual consistency*. The concepts presented within these two models are essential to the following chapters. For every model, we show its specified guarantees to provide consistency at the system side. Eventually, and considering the strength level of these guarantees, we discuss the potential solutions to manage updates conflict situations and further guarantees at the *client side* in some cases. The consistency models are then summarized in Table 3.1.

3.2.1 Strong Consistency

In traditional distributed storage and database systems, the instinctive and correct way to handle replica consistency was to insure a strong consistency state of all replicas in the system at all time. For example, the RDBMs were based on ACID semantics. These semantics are well defined and insure a strong consistency behavior of the RDBM based on the atomicity and consistency properties. Similarly, the POSIX standard for file systems implies that data replicated in the system should always be consistent. Strong consistency guarantees that all replicas must be in a consistent state immediately after an update, before it returns a success. In a perfect world, such semantics and a strong consistency model are the properties that every storage system should adopt. However, insuring strong consistency requires mechanisms that are very costly in terms of performance and availability and limit the system scalability. This was not an issue in the early years of distributed storage systems as the scale and the performance needed at the time were not as important. However, in the era of Big Data and Cloud Computing, this consistency model can be penalizing, in particular if such a strong consistency is actually not required by the applications.

Several models and correctness conditions to insure strong data consistency were proposed over the years. Two of the most common models to provide strong consistency guarantees are *serializability* [30] and *linearizability* [73] that introduce specific constraints on the ordering of the access operations execution.

Given the strength of strong consistency models, no further guarantees are required at the client side. All situations that lead to updates conflict or inconsistency are efficiently handled at the system-side (with eventually performance overhead).

3.2.2 Weak Consistency

For historical purposes, we present the weak consistency model. Weak Consistency was one of the first intuitions that have aimed at relaxing consistency guarantees. The implementation of strong consistency models imposes, in many cases, limitations in both system's design choices and application performance. To overcome these limitations, *Dubois et al.* [53] first introduced the weak ordering model that relaxes the strong guarantees for enhanced performance.

System-side guarantees

Data accesses (read and write operations) are considered as weakly ordered if they satisfy the following three conditions:

- All accesses to a shared synchronization variables are strongly (sequentially) ordered. All processes perceive the same order of operations.
- Data accesses to a synchronization variable are not issued by processors before all previous accesses have been globally performed.
- A global data access is not allowed by processors until a previous access to synchronization variable is globally performed.

From these three conditions, the order of read and write operations, outside critical sections (synchronization variables), can be seen in different orders by different processes as long as they don't violate the aforementioned conditions. However, in [112] [5], it has been argued that not all the three conditions are necessary to reach the intuitive goals of weak ordering. Numerous variation models have been proposed since. *Bisiani et al.* [32] proposed an implementation of weak consistency on distributed memory systems. Timestamps are used to achieve a weak ordering of the operations. A synchronization operation is completed only after all previous operations in the systems reach a completion state.

Client-side guarantees

The following client-side models are weak consistency models, but provide further guarantees to the client.

Read-your-writes. This model guarantees that a process that commits an update will always be able to see its most recent committed update when a read is issued. This might be an important consistency property to provide with weakly ordered systems for a large class of applications. As will be seen further in this section, this is a special case of causal consistency.

Session consistency. Read-your-writes consistency is guaranteed in the context of a *session* (which is a sequence of accesses to data, usually with an explicit beginning and ending). As long as the users access data during the same session, they are guaranteed to access their latest updates. However, the read-your-writes property is not guaranteed to be spanned over different sessions.

Monotonic reads. A process should never read a data item value older than what it has read before. This consistency guarantees that a process's successive reads return always the same value or a more recent one than the previous read.

Monotonic writes. This property guarantees the serialization of the writes by one process. A write operation on a data object or item must be completed before any successive writes by the same process. Systems that do not guarantee this property are notoriously hard to program [124].

3.2.3 Eventual Consistency

In this part we present the eventual consistency model. This model has become very popular in the recent years and is at the center of this Ph.D contributions.

In a replicated storage system, the consistency level defines the behavior of divergence of replicas of logical objects in the presence of updates [115]. Eventual consistency [124, 115, 119] is the weakest consistency level that guarantees convergence. In the absence of updates, data in all replicas will gradually and *eventually* become consistent.

System-side guarantees

Eventual consistency ensures the convergence of all replicas in systems that implement lazy, update-anywhere or optimistic replication strategies [110]. For such systems, updates can be

performed on any replica hosted on different nodes. The update propagation is done in a lazy fashion. Moreover, this update propagation process may encounter even more delays considering cases where network latency is of a high order such as for geo-replication. Eventual consistency is ensured through mechanisms that will guarantee the propagation process will successfully terminate at a future (maybe unknown) time. Furthermore, Vogels [124] judges that, if no failures occur, the size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas in the system.

Eventual consistency by the mean of lazy asynchronous replication may allow better performance and faster accesses to data. Every client can read data from local replicas located in a geographically close datacenter. However, if an update is performed on one of the replicas and is yet to be propagated to others because of the asynchronous replication mechanism, a client reading from a distant replica may read stale data.

Further client–side guarantees, as introduced with weak consistency, might be provided at the level of client processes in order to cope with the various forms of undesirable inconsistency for user' applications.

Handling of update conflicts

Eventual consistency is most suitable for a given class of applications, for which update conflicts are rare and stale data are, in most, harmless. In contrast, the management of update conflicts may present serious problems for other types of applications. In [119], two examples that illustrate the typical use case and show the potential gains with this consistency model are presented. The worldwide domain name system (DNS) is a perfect example of a system for which eventual consistency is the best fit. The DNS namespace is partitioned into domains where each domain is assigned to a naming authority. This is an entity that will be responsible for this domain and is the only one that can update it. This scheme eliminates the update-update conflict. Therefore, only the read-update conflict needs to be handled. As updates are less frequent, in order to maintain system availability and fast accesses for users read operations, lazy replication is the best–fit solution. Another example is the World Wide Web. In general, each web page is updated by a single authority, its Webmaster. This also avoids any update-update conflict. However, in order to improve performance and lower read access latency, browsers and Web proxies are often configured to keep a fetched page in a local cache for future requests. As a result, a stale out-of-date page may be read. However, many users find this inconsistency acceptable (to a certain degree) [119].

Within many applications, update conflicts may not need a strict ordering between them but can be handled based on the application semantics instead. A famous example of this case is the Amazon user *shop cart* application where updates can be easily merged no matter their order of occurrence. In this context, eventually consistent systems provide, generally, mechanisms to handle update conflicts.

3.2.4 Causal Consistency

Causal consistency is another model that relaxes strong consistency ordering as to reduce the performance overhead. It utilizes the causal relation between operations in order to provide minimum causality guarantees.

System-side guarantees

Causal consistency is a consistency model where a sequential ordering is always preserved only between operations that have causal relation. In [6][91], two operations a and b have a potential causality if one of the two following conditions are met: a and b are executed in a single thread and one operation execution precedes the other in time; or if b reads a value that is written by a. Moreover, a causality relation is transitive. If a and b have a causal relation, b and c have a causal relation as well, then a and c have a causal relation. Operations that execute concurrently do not share a causality relation. Therefore, causal consistency does not order concurrent operations.

Handling of update conflicts

In [91], a model that combines causal consistency and convergent conflict handling is presented and called causal+. Since concurrent operations are not ordered by causal consistency, two writes to the same key or data object would lead to a concurrent update conflict where different data changes may be performed at the level of at least two different replicas. The main challenge for such situations is to ensure replica convergence to the same state. In contrast to eventual consistency, the update conflict is a direct result of performing updates concurrently. With eventual consistency however, a conflict may occur with two updates to the same data but not at the same time. The convergent conflict handling aims at handling all the replicas in the same manner using a handler function in order to ensure that all replicas will be in the same state at a latter time. To reach convergence, all conflicting replicas should consent to an agreement. Various conflict handling methods were proposed such as last-writer-wins rule [121], through user intervention, or using versioning mechanisms that allow merging of different versions in one as in Amazon's eventually consistent Dynamo storage system. In this context, it has been shown that implementing causal consistency with last-writer-wins rule to handle updates conflicts at wide scales provides performance comparable to that of eventually consistent systems [92].

3.2.5 Timeline Consistency

The timeline consistency model was proposed specifically for the design of Yahoo! PNUTS [44], the storage system designed for Yahoo! Web applications. This consistency model was proposed to overcome the inefficiency of serializability of transactions at massive scales and geo-replication. Moreover, it aims to limit the weaknesses of eventual consistency.

System-side guarantees

Transaction serializability was avoided as a design choice within Yahoo! PNUTS. This was mainly due to the observation that web applications typically manipulate one record at a time. Therefore, a per-record timeline consistency was introduced. Unlike eventual consistency, where operations order can vary from one replica to another, all replicas of a record perform the operations in the same "correct" order. For instance, if two concurrent updates are performed, all replicas will execute them in the same order and thereby avoid inconsistencies. Nevertheless, data propagation to replicas is done lazily, which makes the conver-

Table 3.1: Consistency Models

Consistency Model		Guarantees
Strong Consistency	serializability	Serial order of concurrent executions of a set of serialization units (set of operations).
	linearizability	Global total order of operations (single operations), every operation is perceived instantaneously.
Weak Consistency	Read-your-writes	A process always sees its latest update on read operations.
	Session consistency	Read-your-writes consistency is guaranteed only within a session.
	Monotonic reads	Successive reads must always return the same or a more recent value than a previous read.
	Monotonic writes	A write operation must always complete before any successive writes.
Causal Consistency		Total ordering between operations that have a causal relation.
Eventual Consistency		In the absence of updates, all replicas will gradually and eventually become consistent.
Timeline Consistency		All replicas perform operations on one record in the same “correct order".

gence of all replicas eventual. This allows clients to read data from local replicas that may be in a stale version. In order to preserve the order of operations for a given record, one replica is designated dynamically as a master replica for the record that handles all the updates.

This model avoids major problems related to update conflicts, that eventually–consistent systems might suffer from, since update operations are all executed at the same order everywhere. Moreover, the model still provides some flexibility to allow read update conflicts and therefore reducing performance overhead caused by synchronization. However, eventual consistency still outperforms timeline consistency by avoiding updates synchronization.

3.2.6 Discussion

Table 3.1 summarizes the presented consistency models. These models provide different levels of guarantees and are designed for different classes of applications and use cases. Strong models such as linearizability and serializability provide the strongest forms of consistency that eliminates operations conflict situations. However, this comes at the cost of performance overhead due to the application of expensive necessary mechanisms in order to deliver such guarantees. In this context, the weak consistency models were introduced to relax the strong guarantees for applications that do not require the strongest forms of consistency. As a result, the performance overhead is significantly reduced. In order to deal with weak guarantees, additional operation ordering requirements can be imposed on the client level as to cope with the consistency needs. In the context of relaxing the strong consistency guarantees, three models have become very popular in recent years. They differ mainly in the way they handle conflicts. *Eventual consistency* is the weakest consistency model that guarantees the convergence of replicas. In this model, update conflicts resolution is postponed to a future time favoring availability of data. *Causal consistency* does not allow the violation of causal

relations between updates. Therefore, only concurrent simultaneous updates can lead to the conflict situations. As opposed to these two models, *timeline consistency* orders updates deterministically and does not allow updates conflicts. Data propagation is however, eventual and thus, read-update conflicts may occur.

While insuring strong consistency comes at the cost of performance overhead and decreased availability, the weaker consistency models might allow far too much inconsistency, in particular eventual consistency. Moreover, for dynamic workloads—which are common for applications running in the cloud such as Web shops that exhibit high load variability—, both the consistency and performance needs might be variable and can change overtime. Therefore, one static model (that can guarantee either strong consistency or weak consistency) is not suitable to satisfy the varying requirements of this type of applications. In this context, the need for adaptive models strongly imposes itself.

3.3 Cloud Storage Systems

To illustrate the previously introduced models with some practical examples of systems, in this section we describe some state-of-the-art cloud storage systems. These systems are adopted by the big cloud vendors, such as Amazon Dynamo, Apache Cassandra, Yahoo! PNUTS, and Google Spanner. We focus on Amazon Dynamo, and Cassandra in particular as two illustrative systems that implement eventual consistency. Moreover, Cassandra is the system used for most of our experimental evaluations in the following chapters. In the following, cloud storage systems are presented by introducing their targeted applications and use cases, their data models, their design principles and adopted consistency, and their APIs. We then, give an overview of more systems and their real-life applications and use-cases in Table 3.2.

3.3.1 Amazon Dynamo

Amazon Dynamo [49] is a storage system designed by Amazon engineers to fit the requirements of their web services. Dynamo provides the storage backend for the highly available worldwide Amazon.com e-commerce platform and overcomes the inefficiency of RDBMs for this type of applications. RDBMs fail over meeting the availability and the response time required by Amazon services due to the adoption of ACID semantics. Reliability and scaling requirements within this platform services are high. Moreover, availability is very important, as the increase of latencies by only minimal fractions can cause financial losses. Dynamo provides a flexible design where services may control availability, consistency, cost-effectiveness and performance tradeoffs. Dynamo's data model relies on a simple key/value scheme. Since the targeted applications and services within Amazon do not require complex querying models, a record-based or key-based queries are considered both enough in term of requirements and efficient in terms of performance scaling.

Dynamo's design relies on a consistent hashing-based partitioning scheme [80]. In the implemented scheme, the resulting range or space of a hash function is considered as a ring. Every member of the ring is a virtual node (host) where a physical node may be responsible for one or more virtual nodes. The introduction of virtual nodes, instead of using fixed physical nodes on the ring, is a choice that provides better availability and load balancing under

failures. Each data item can be assigned to a node on the ring based on its key. The hashed value of the key determines its position on the ring. Then, data is assigned to the closest node on the ring clockwise. Moreover, data is replicated on the successive $K-1$ nodes for a given replication factor K, avoiding virtual nodes that belong to the same physical nodes. All the nodes on Dynamo are considered equal and are able to compute the *reference list* for any given key. The *reference list* is the list of nodes that store a copy of the data referenced by that key.

Dynamo is an eventually–consistent system. Updates are asynchronously propagated to replicas. As data is usually available while updates are being propagated, clients may perform updates on older versions of data for which the last updates have not been committed yet. As a result, the system may suffer from update conflicts. To deal with these situations, Dynamo relies on data versioning. Every updated replica is assigned a new immutable version. The conflicting versions of data resulting from concurrent updates may be solved at a latter time. This allows the system to be always available and fast to respond to client requests. Versions that share a causal relation are easy to solve by the system based on syntactic reconciliation. However, a difficulty arises with versions branching. This often happens in the presence of failures combined with concurrent updates and results in conflicting versions of data. The reconciliation in this case is left to the client rather than to the system because the latter lacks the semantic context. The reconciliation is performed by collapsing the multiple data versions into one (semantic reconciliation). A simple example is the case of the *shopping cart* application. This application chooses to merge the diverging versions as a reconciliation strategy.

Clients can interact with dynamo through a flexible API that provides various consistency configurations. Replica consistency is handled by a quorum-like system. In a system that maintains N replicas, R is the minimum number of nodes (replicas) that must participate in the read operation, and W is the minimum number of nodes that must participate in the write operation are configured on a per operation basis and are of high importance. By setting these two parameters, one can define the tradeoff between consistency and latency. A configuration that provides $R + W > N$ is a quorum-like setting. This configuration insures that the last up-to-date replica is included in the quorum and thus in the response. However, the operation latencies are as small as the longest replica response time. In a configuration where $R + W < N$, clients may be exposed to stale versions of data.

3.3.2 Cassandra

Many web services and social networks are data-intensive and deal with the problem of data deluge. The Facebook social networking platform is the largest networking platform serving hundred millions of users at peak times and having no less than 900 million active users [59]. Therefore, and in order to keep users satisfied within such services, an efficient Big Data management that guarantees high availability, performance, and reliability is required. Moreover, a storage system that fulfills these needs must be able to elastically scale out to meet the continuous growth of the data-intensive platform. Cassandra [85] is a highly available, highly scalable, distributed storage system that was first built within Facebook. It was designed for managing large objects of structured data spread over a large amount of commodity hardware located in different datacenters worldwide.

The design of Cassandra was highly inspired by that of two other distributed storage

systems. Implementation choices and consistency management are very similar to the ones of Amazon Dynamo (except for in-memory management) while its data model is derived from that of Google BigTable [40] model. Data are stored in tables and indexed by row keys. Moreover, this model is *column family* based. The column keys are grouped into sets called *column families*. For each table, column families are defined and column keys within a column family can be created dynamically. Every operation on a single row key is atomic per replica without considering which columns are accessed. Such a data model provides great abilities for structured large data, as it offers a more flexible yet efficient data access. Moreover, it provides an efficient dynamic memory management (due to column families and their related data structures). Like BigTable, Cassandra keeps data in-memory in small tables called *memtables*. When a memtable size grows over a given threshold, it is considered as full and data is flushed into an *sstable* that will be dumped to the disk.

The Cassandra partitioning scheme is based on consistent hashing. Unlike Dynamo, which uses virtual nodes to overcome the non-uniformity of load distribution, every node on the ring is a physical host. Therefore, and in order to guarantee uniform load distribution, Cassandra uses the same technique as in [116], where lightly–loaded nodes move on the ring. Replication in Cassandra is performed in the same manner as in Dynamo. Moreover, Cassandra implements few replication strategies that consider the system topology. Therefore, strategies that are *Rack UnAware*, *Rack Aware*, and *Datacenter Aware* are provided. For the two latter strategies, Cassandra implements algorithms in Zookeeper [75] in order to compute *the reference list* for a given key. This list is cached locally at the level of every node as to preserve the zero-hop property of the system.

Similar to Dynamo, Cassandra provides a flexible API to clients. In this context, various consistency levels [4] are proposed per operation. A write of consistency level *One* implies that data has to be written to the commit log and memory table of at least one replica before returning a success. Moreover, as shown in Figure 3.2, a read operation with consistency level *All* (strong consistency) implies that the read operation must wait for all the replicas to reply and insures that all replicas are consistent in order to return the data to the client. In contrast, in a read consistency of level of *Quorum* (Quorum is computed as: $\lfloor \frac{replication\,factor}{2} + 1 \rfloor$), two out of the three (when the replication factor is set to three) replicas are contacted to fulfill the read request, and the replica with the most recent version would return the requested data. In the background, a read repair will be issued to the third replica and will check for consistency with the first two. If inconsistency occurs, an asynchronous process will be launched to repair the stale nodes at a latter time.

3.3.3 Yahoo! PNUTS

PNUTS [44] is a massively parallel geographically distributed storage system. Its main purpose is to host and serve data for Yahoo! Web applications. Its design and implementation were driven by Yahoo!'s requirements for a data management platform that provides scalability, fast response, reliability, and high availability in different geographical areas. PNUTS relies on a novel relaxed consistency model to cope with availability and fault-tolerance requirements at large scale. In this context, it provides the user with a simplified relational model. Data is stored in a set of tables of records with multiple attributes. An additional data type provided to the users is the “blob" type. A blob encapsulates arbitrary data structures (not necessarily large objects) inside records.

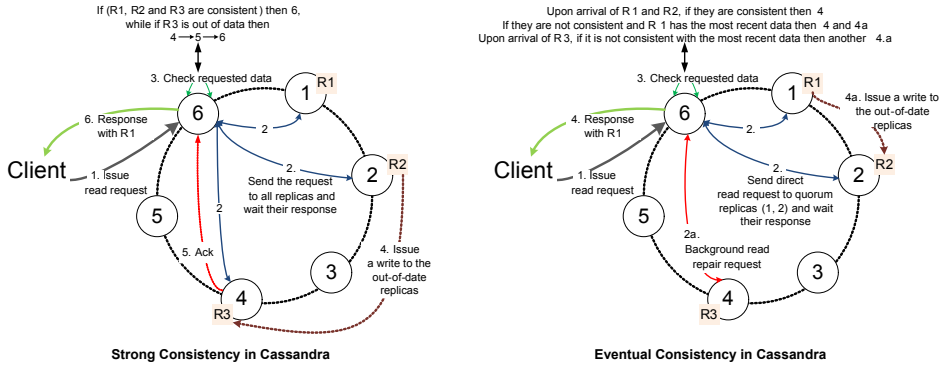


Figure 3.2: Consistency in Cassandra

PNUTS divides the system into a set of regions. Regions are typically, but not necessarily, geographically distributed. Every region consists of a set of *storage units*, a *tablet controller* and a set of *routers*. Data tables are decomposed horizontally into smaller data structures called *tablets* that are stored across storage units (servers) within multiple regions. On the other hand, the routers functionality is to locate data within tablets and storage units based on a *mapping* computed and provided by the tablet controller. PNUTS introduces the novel consistency model of *per-record timeline consistency* described in the consistency models section. Therefore, it uses an asynchronous replication scheme. In order to provide reliability and replication, PNUTS relies on a pub/sub mechanism, called *Yahoo! Message Broker* (YMB). With YMB, PNUTS avoids other asynchronous replication protocols such as Gossip, and optimizes geographical replication. Moreover, a replica does not need to acquire the location of other replicas. Instead, it needs just to subscribe to the data updates within YMB.

In order for applications and users to deal with timeline consistency, API calls which provide varying consistency guarantees were proposed. The *read-any* call may return stale data to the users favoring performance and fast response to consistency. In common cases, a class of applications requires the read data to be more recent than a given version. The API call *read-critical (required_version)* is proposed to deal with these requirements. In contrast, the *read-latest* call always returns the most recent version of data. This call however may be costly in terms of latency. Moreover, the API provides two calls for writing data. The *write* call gives ACID guarantees for the write (a write is a transaction with a single operation). In contrast, *test-and-set-write(required_version)* checks the version of the actual data in the system. If, and only if, the version matches *required_version*, the write is performed. This flexible API calls give a degree of freedom to applications and users to choose their consistency guarantees and control their availability, consistency, and performance tradeoffs.

3.3.4 Google Spanner

Spanner [45] is a scalable, globally distributed database that provides synchronous replication and ensures strong consistency. While many applications within Google require geo-replication for global availability and geographical locality reasons, a large class of these applications still needs strong consistency and an SQL-like query model. Google BigTable[40] still serves and manages data efficiently for many applications, but it only guarantees eventual consistency at global scale and provides a NoSQL API. Therefore, Spanner is designed to

Table 3.2: Cloud Storage Systems

Storage System	**Consistency Model**	**Data Model**	**Cloud Applications/Services**	**API**
Amazon Dynamo	Eventual Consistency	key/value	Amazon.com e-commerce platform, Few AWS (Amazon Web Services) (eg. DynamoDB)	multiple consistency levels
Cassandra	Eventual Consistency	column families	Facebook inbox search, Twitter, Netflix, eBay, SOUNDCLOUD, RackSpace Cloud	multiple consistency levels
Riak [109]	Eventual Consistency	key/value	Yammer private social network, Clipboard, GitHub, enStratus Cloud	multiple consistency levels
Voldemort [125]	Eventual Consistency	key/value	LinkedIn, eHarmony, GiltGroup, Nokia	multiple consistency levels
CouchDB [19]	Eventual Consistency	document-oriented	Ubuntu One cloud, BBC (Dynamic Content Platform), Credit Suisse (Market Place Framework)	RESTful API
MongoDB [96]	Eventual Consistency	document-oriented	SAP AG Software Entreprise, MTV, and Sourceforge	CRUD API
Yahoo PNUTS!	Timeline Consistency	relational-like	Yahoo web applicastions	multiple consistency guarantees
Google BigTable [40]	Strong Consistency	column families	Google analytics, Google earth, Google personalized search	NoSQL API
Google Megastore [26]	Strong Consistency	semi-relational	Google applications: Gmail, Picasa, Google Calendar, Android Market, and AppEngine	SQL-like
Google Spanner	Strong Consistency	semi-relational	Google F1	SQL-like
Redis [107]	Strong Consistency	key/value	Instagram, Flickr, The guardian news paper	NoSQL API
Microsoft Azure Storage [37]	Strong Consistency	blob tables	Microsoft internal applications: networking search, serving video, music and game content, Blob storage cloud service	RESTful API
Apache HBase [21]	Strong Consistency	column families	Facebook messaging system, traditionally used with Hadoop for large set of applications	NoSQL API

overcome BigTable insufficiencies for the aforementioned class of applications and provides globe scale external consistency (linearizability) and SQL-like query language similar to that of Google Megastore [26]. Data is stored into semi-relational tables to support an SQL-like query language and general-purpose transactions.

The architecture of Spanner consists of a *universe* that may contain several *zones* where zones are the unit of administrative deployment. A zone additionally presents a location where data may be replicated. Each zone encapsulates a set of *spanservers* that host data tables which are split into data structures called *tablets*. Spanner timestamps data in or-

der to provide multi-versioning features. A *zonemaster* is responsible for assigning data to spanservers whereas, the *location proxies* components provide clients with information to locate the spanserver responsible for its data. Moreover, Spanner introduces an additional data abstraction called *directories*, which are a kind of buckets to gather data that have the same access properties. The directory abstraction is the unit used to perform and optimize data movement and location. Replication is supported by implementing a Paxos protocol. Each spanserver associates a Paxos state machine with a tablet. The set of replicas for a given tablet is called a *Paxos group*. For each tablet and its replicas, a long-lived Paxos leader is designated with a time-based leader lease. The Paxos state machines are used to keep a consistent state of replicas. Therefore, writes must all initiate the Paxos protocol at the level of the Paxos leader while reads can access Paxos states at any replica that is sufficiently up-to-date. At the level of the leader replica, a lock table is used to manage concurrency control based on a two-phase locking (2PL) protocol. Consequently, all operations that require synchronization should acquire locks at the lock table.

In order to manage the global ordering and external consistency, Spanner relies on a time API called *TrueTime*. This API exposes clock uncertainty and allows Spanner to assign globally meaningful commit timestamps. The clock uncertainty is kept small within the TrueTime API relying on atomic clocks and GPS based clocks at the level of every datacenter. Moreover, when uncertainty grows to a large value, Spanner slows down to wait out that uncertainty. The TrueTime API is then used to guarantee spanner desired correctness properties for concurrent executions. Therefore, providing external consistency (linearizability) while enabling lock-free read-only transactions and non-blocking reads in the past.

Spanner presents a novel globally-distributed architecture that implements the first globally ordered system with external consistency guarantees. While such guarantees were estimated to be fundamental for many applications within Google, it is unclear how such an implementation affects latency, performance, and availability. In particular, the write throughput might suffer from the two-phase locking mechanism, which is known to be very expensive at wide scale. Moreover, it is not obvious how Spanner deals with availability during network failures.

3.3.5 Discussion

As Cloud Computing technology emerges, more and more cloud storage systems have been developed. Table 3.2 gives an overview of the four aforementioned cloud storage systems along with several other storage system examples. Many systems implement eventual consistency. These systems are usually destined to serve social networks, web shop applications, document-based applications, and cloud services. Commonly, they adopt one of the following data models: key/value, column families, and document-oriented. Moreover, many of these systems provide the user with a flexible API that offers various consistency levels. On the opposite side, systems that implement strong consistency serve many applications including services such as mail service, advertisement, image hosting platforms, data analytics applications, and few cloud services as well. These applications, in general, require strong consistency while their availability and performance requirements are not as high as web shop services for instance. These systems implement, generally, a semi-relational data model, column families, and rarely, a key/value model. Moreover, they usually provide users with SQL-like API.

Many of these systems, such as Google Spanner or MongoDB, are designed for a particular type of applications, and therefore the consistency model is chosen based on that. In this context, such systems are not suited for all types of workloads and applications and thus, might fail in meeting consistency and performance requirements. On the other hand, general–purpose systems such as Cassandra provide flexible APIs and different consistency levels for runtime usage. However, selecting its consistency level at runtime is difficult and lacks automation. When choosing a consistency level, it is not obvious what are the consequences in terms of inconsistency and performance overhead. For this type of systems, automated tools that help evaluate the consistency guarantees and the provided performance of a given consistency level and subsequently selects the pertinent level to suit the application requirements are needed.

3.4 Adaptive Consistency

A wide range of applications either require a strictly strong form of consistency or settle for only static eventual consistency. However, for another class of applications, consistency requirements are not obvious as they depend on data access behavior dynamicity, client needs, and the consequences (or the cost) of reading inconsistent data. Typical applications that fall in this class include auction systems and web shop applications. For these types of applications, availability and fast accesses are vital. Therefore, strong consistency mechanisms may be too costly. While high levels of consistency are strongly desired for these particular applications, they are not always required. When an auction starts or in the not so busy periods of a web shop, a weaker form of consistency is sufficient and do not cause anomalies that the storage system can not handle. However, strong consistency is required towards the end of the auction as well as in the busy holiday periods, as heavy accesses are expected and data inconsistency might be of disastrous consequences. As with this type of situations, static eventual or strong forms of consistency lead both to undesirable consequences.

In order to cope with the dynamicity of accesses behavior at the massive cloud scale, various adaptive and dynamic consistency approaches were introduced. Their goal is to use strong consistency only when it is necessary. These approaches differ in the target consistency tradeoffs (*e.g.*, consistency-cost in [82] and consistency-performance in [87, 90]) and in the way they define the consistency requirements. Hereafter, we present two adaptive consistency models (next section will be devoted for our adaptive consistency solution Harmony). For both models, we start by presenting the motivation and the use case, then we present the adaptive approach, and finally we describe the model implementation on the targeted infrastructure.

3.4.1 RedBlue Consistency

Due to the high network latencies, strong consistency guarantees are too expensive when storage systems are geographically distributed. Therefore, weaker consistency semantics such as eventual consistency is the most popular choice for applications that require high availability and performance. However, weaker consistency models are not suitable for all applications classes, even if most operations within one application require only eventual consistency. For instance, in the case of a social network, a transaction that might combine

privacy-related updates among other social activity operations might need more than eventual consistency. Privacy-related operations require strong ordering at all geographical sites in order not to violate the privacy setting of the user. On the other hand, social activity operations might only require eventual convergence of replicas no matter the ordering of the operations.

RedBlue consistency [87] was introduced in order to provide as fast responses as possible and consistency when necessary. It provides two types of operations: Red and Blue. Blue operations are executed locally and replicated lazily. Therefore, their ordering can vary from site to site. In contrast, Red operations require a stronger consistency. They must satisfy serializable ordering with each other and as a result generate communication across sites for coordination. Subsequently, the RedBlue order is defined as a partial ordering for which all Red operations are totally ordered. Moreover, every site has a local causal serialization that provides a total ordering of operations that are applied locally. This definition of the RedBlue consistency does not guarantee the convergence of the replicas state. Convergence is reached if all causal serializations of operations at the level of each site reach the same state. However, with the RedBlue consistency, Blue operations might have different orders in different sites. Therefore, non-commutative operations executed in a different order won't allow replicas convergence. As a result, non-commutative operations should not be tagged as Blue if convergence is to be insured. An extension of the RedBlue consistency consists in splitting original application operations into two components. A *generator operation* that has no side effect and is executed only at the primary site and *shadow operation*, which is executed at every site. *Shadow operations* that are non-commutative or violate the application variant (*e.g.*, negative values for a positive variable) are labeled Red while all other *shadow operations* are labeled blue.

The RedBlue consistency is implemented in a storage system called *Gemini* [87]. Gemini uses MySQL as its storage backend. Its deployment consists of several sites where each site is composed of four components: a *storage engine*, a *proxy server*, *concurrency coordinator*, and *data writer*. The proxy server is the component that processes client requests for data hosted on the storage engine (a relational database). *Generator operation*s are performed on a temporary private scratchpad, resulting in a virtual private copy of the service state. Upon the completion of a *generator operation*, the proxy server sends the *shadow operation* to the concurrency coordinator. The latter notifies the proxy server whether the operation is accepted or rejected according to the RedBlue consistency. If accepted, the operation is then delegated to the local data writer in order to be executed in the storage engine.

RedBlue consistency provides the so needed adaptivity for systems that require performance while do not require a strictly strong consistency. However, one difficulty that arises is the categorization of operations in Red and Blue. It is difficult to intuitively define for each operation its color without a huge analysis effort. Furthermore, and considering that target applications are large–scale applications and geographically distributed, it is fair to assume a high probability for them to operate on large volumes of data and various data types. This in turn, makes the categorization of operations on data extremely difficult showing a big necessity for automation.

3.4.2 Consistency Rationing

Data created and processed by the same application might be different, and so the consistency requirements on them. For instance, data processed by a web shop service can be of different kinds. Data kinds may include customers profiles and credit card information, product sold data, user preferences etc. Not all these data kinds have the same requirements in terms of consistency and availability. Moreover, within the same category, data might exhibit dynamic and changing consistency requirements. As an example, an auction system data might require lower levels of consistency at the start of the auction than towards the end of it.

The *consistency rationing* model [82] allows designers to define consistency requirements on data instead of transactions. It divides data into three categories: *A*, *B*, and *C*. Category *A* data requires strong consistency guarantees. Therefore, all transactions on this data are serializable. However, serializability requires protocols and implementation techniques as well as coordination, which are expensive in terms of monetary cost and performance. Data within *C* category is data for which temporary inconsistency is acceptable. Subsequently, only weaker consistency guarantees, in the form of session consistency, are implemented for this category. This comes at a cheaper cost per transaction and allows better availability. The *B* category on the other hand presents data for which consistency requirements change in time as in the case for many applications. These data endure adaptive consistency that switch between serializability and session consistency at runtime whenever necessary. The goal of the adaptive consistency strategies is to minimize the overall cost of the provided service in the cloud. The general policy is an adaptive consistency model that relies on the probability of update conflicts. It observes the data access frequency to data items in order to compute the probability of access conflicts. When this probability grows over an adaptive threshold, serializability is selected. The computation of the adaptive threshold is based on the monetary cost of weak and strong consistency, and the expected cost of violating consistency.

Consistency rationing is implemented in a system that provides storage on top of *Amazon Simple Storage Service* (S3) [13], which provides only eventual consistency. Clients Requests are directed to application servers. These servers are hosts on *Amazon EC2* [9]. Therefore, application servers interact with the persistent storage on *Amazon S3*. In order to provide consistency guarantees, the update requests are buffered in queues called pending updates queues that are implemented on the *Amazon Simple Queue Service* (SQS) [12]. Session consistency is provided by always routing requests from the same client to the same server within a session. In contrast, and in order to provide serializability, a two-phase locking protocol is used.

Consistency rationing provides a high level of adaptiveness for applications in the cloud. The categorization of data into three categories allows defining the consistency requirements with a high precision. Moreover, applications that process data in the *B* category can benefit from adapting consistency according to the access pattern dynamicity. However, the data categorization within consistency rationing lacks automation. This presents a huge obstacle for Big Data applications that process large volumes of data with various data types. Defining which data belong to which category manually becomes extremely difficult at such scales.

3.5 Summary

This chapter discusses a major open issue in cloud storage systems: the management of consistency for replicated data. Despite a plethora of cloud storage systems available today, data consistency schemes are still far from satisfactory. We take this opportunity to ponder the *CAP* theorem 13 years after its formulation and discuss its implications in the modern context of Cloud Computing. The tension between Consistency, Availability and Partition Tolerance has been handled in various ways in existing distributed storage systems (e.g., by relaxing consistency at wide-area level). We therefore provide an overview of the major consistency models and approaches used for providing scalable, yet highly available services on clouds. We categorize the consistency models according to their consistency guarantees into: (1) strong form of consistency including linearizability and serializability, and (2) weaker form of consistency including eventual, causal and timeline consistency. For the weaker consistency models, we elaborate on what additional operation ordering is applied to handle conflict situations. Cloud storage is foundational to Cloud Computing because it provides a backend for hosting not only user data but also the system-level data needed by cloud services. We survey the state-of-the-art cloud storage systems used by the main cloud vendors (*i.e.*, in Amazon, Google, and Facebook). In addition to a general presentation of these systems architectures and use cases, we discuss the employed consistency model by each cloud storage system. The survey helps us to understand the mapping between the applied consistency technique and target requirements of the applications using these cloud solutions. Moreover, and in order to handle Big Data, the scale of cloud systems is extremely increasing and the cloud applications are significantly diversifying (*e.g.*, access pattern and diurnal/monthly loads). We advocate self-adaptivity as a key means to approach the tradeoffs that must be handled by the user applications. We review several approaches of adaptive consistency that provide flexible consistency management for users to reduce performance overhead when data are distributed across geographically distributed sites.

Part II

Contributions: Adaptive Consistency Approaches for Cloud Computing

Chapter 4

Consistency vs. Performance: Automated Self-Adaptive Consistency in the Cloud

Contents

This chapter is mainly extracted from the paper: *Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage.* Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, Maria S. Pérez. In the proceeding of the 2012 IEEE International Conference on Cluster Computing (Cluster 2012), Beijing, China, September 2012, pp.293-301

CONSISTENCY management within cloud storage systems is of high importance. As highlighted in the previous chapter, multiple consistency tradeoffs impose themselves. The Consistency–Performance tradeoff is, arguably, the main tradeoff. Many static consistency solutions fail in reaching an efficient equilibrium between consistency and performance for dynamic cloud workloads. In this context, more opportunistic, adaptive consistency models are needed in order to meet the requirements (and the Service Level Agreements SLAs) of Big Data applications. However, most of the existing adaptive policies either lack automation (necessary to select the pertinent consistency level for access operations of large–scale applications) or fail to apprehend and include the specific consistency requirements of the application outside its access pattern. Hereafter, in this chapter, we tackle this specific issue of consistency management for dynamic workloads in the cloud. Accordingly, we provide an adaptive model that tunes the consistency level at runtime in order to provide consistency when needed and performance when possible.

4.1 Motivation

Replication is an important feature for Big Data in cloud storage in order to deal with many challenges as highlighted in Chapter 2. However, with replication, consistency comes into question. In this context, eventual consistency has become very popular alternative to strong consistency as to cope with performance and availability requirements of cloud and Big Data systems. By avoiding synchronous replication, eventually–consistent systems provide low operation latencies and high levels of availability that are critical for many applications. For instance, the cost of a single hour of downtime for a system doing credit card sales authorizations has been estimated to be between $2.2M-$3.1M [106].

As seen in the previous chapter, many cloud storage systems have been developed such as Amazon Dynamo [49], Cassandra [85], Google BigTable [40], Yahoo! PNUTS [44], and HBase [21]. These solutions are practical to use as cloud and web service storage backend. They allow many web services to scale up their systems in an extreme way, while maintaining performance with very high availability. For example, Facebook uses Cassandra to scale up to host and serve *inbox* data for more than 800 million active users [59]. However, the undoubted availability and performance of such solutions prove to be too costly in terms of inconsistency. As shown in [126], under heavy reads and writes some of these systems may return up to 66.61 % stale reads. This is an *alarming* rate, meaning that most probably two out of three reads are "useless".

In this chapter, we address the tradeoffs between consistency and performance on the one hand, and consistency and availability on the other. Accordingly, we propose an automated and self-adaptive approach, named *Harmony*, that tunes the consistency level at runtime to reduce the probability of stale reads caused by the dynamicity of cloud systems (i.e., the network latency which directly affects updates propagation to replicas) and the application demands (i.e., the frequency of access patterns during reads, writes and updates), thus providing adequate tradeoffs between consistency and both performance and availability. *Harmony* embraces an intelligent estimation model to automatically identify the key parameter affecting the stale reads such as the system states (network latency) and application requirements. *Harmony*, therefore, elastically scales up/down the number of replicas involved in read operations to maintain a low (possibly zero) tolerable fraction of stale reads,

hence improving the performance of the applications while meeting the desired consistency level.

4.2 *Harmony*: Elastic Adaptive Consistency Model

4.2.1 Zoom on Eventual Consistency Levels in Cloud Storage

The way consistency is handled has a big impact on performance. Traditional synchronous replication (strong consistency) dictates that an update must be propagated to all the replicas before returning a success. In contrast, eventual consistency by means of asynchronous quorum replication propagates data lazily to other replicas. Here the consistency level is commonly chosen on a per-operation basis and is represented by the number of replicas in the quorum (a subset of all the replicas). A quorum is computed as: $\lfloor (replication\ factor/2) + 1 \rfloor$. Data accesses and updates are performed to all replicas in the quorum. Thus, using this level for both read and write operations guarantees that the intersection of replicas involved in both operations contains at least one replica with the latest update. A partial quorum has a smaller subset of replicas, hence returning the most recent data when read is issued, is not guaranteed. In the following, the term *consistency level* refers to the number of replicas involved in the access operation (read or write).

As presented in Chapter 3, many cloud storage systems such as Dynamo [49], Cassandra [85], Voledemort [125], and Riak [109] adopt asynchronous quorum replication [72, 64]. This gives the application writer more flexibility when selecting the type of consistency that is appropriate for each operation. This is a useful feature, but until now no automatic adaptive model has been proposed for these systems. This means that the application writer has to choose the type of consistency for every operation, which is extremely difficult when no information is available regarding the read and write frequencies, network latency, and the system state in general, or when operating on a very large scale. We present *Harmony*, an approach which aims to make this task automatic for the operations that are not critical or do not need a strictly strong consistency. This is achieved using just a small hint about the application needs.

4.2.2 *Harmony*

The goal of *Harmony* is to dynamically and elastically handle consistency at run time, in order to provide adequate tradeoffs between consistency and both performance and availability. Accordingly, *Harmony* considers not only the application requirements but also the storage system state. Moreover, rather than relying on a standard model based only on the access pattern to define the consistency requirement of an application – which is the case for most existing work – *Harmony*, in addition, uses the *stale read rate* of the application to precisely define such a requirement.

Why use the stale reads rate to define the consistency requirements of an application?

As an example, we consider two applications that may have at some point the same access pattern. One is a web shop application that can have heavy reads and writes during the busy holiday periods, and a social network application that can also have heavy access during important events or in the evening of a working day. These two applications may have the same behavior at some point and are the same from the point of view of the system when monitoring data accesses and network state as well, thus they may be given the same consistency level. However, the consequences of stale reads are not the same for both applications. A social network application can tolerate a higher number of stale reads than a web shop application: a stale read has little effects on the former, whereas it could result in anomalies for the latter. Consequently, defining the consistency level in accordance to the stale reads rate can precisely reflect the application requirement.

We propose the following model for distributed storage systems. In our context, data may be replicated over geographically distant data centers. In order to predict the effect of weaker consistencies, we compute θ_{stale}, the estimation of the stale read rate in the system. The consistency requirement of an application should be determined by providing the rate of reads that should be fresh; in other words, the rate of stale reads that is tolerated by the application. Let this be app_stale_rate. For critical applications that require strong consistency, this rate should be 0%. Similarly, an application that does not need any consistency at all, such as an application that consists of only reads from archives, the rate could be 100% (which corresponds to static eventual consistency). A naïve way to map the consistency requirements of an application to the app_stale_rate is the following: for an application that needs an average consistency, the rate should be 50%. An application that needs less than average consistency should have a rate of 75%, and an application that requires higher consistency should use 25%. This rate is tunable and can be defined by studying the behavior and the semantics of an application.

Additionally, in the case of distributed data replication, network latency may be high and thus, a performance-defining factor. Other than app_stale_rate, in our model, we consider the network latency and the application access pattern. We permanently collect such information in order to estimate the stale read rate. From a higher level perspective, our solution uses the following decision scheme shown in Algorithm 1 :

Algorithm 1: *Harmony*: consistency–tuning algorithm

Input: app_stale_rate is the pre-defined consistency requirement.
Description: θ_{stale} is estimated stale reads rate.
Output: The most pertinent Consistency Level
if $app\_stale\_rate \geq \theta_{stale}$ **then**
  Choose eventual consistency (Consistency Level = One)
else
  Compute X_n the number of always consistent replicas necessary to have $app\_stale\_rate \geq \theta_{stale}$
  Choose consistency level based on X_n
end

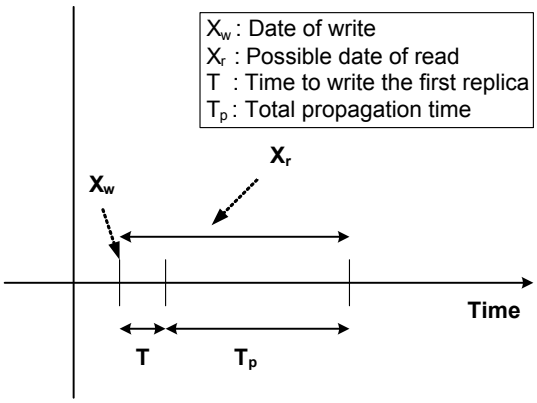


Figure 4.1: Situation that leads to a stale read

The default consistency level is the basic eventual consistency that allows reading from only one replica. When such a level may not satisfy the consistency requirements of an application due to the growing number of stale reads, the number of replicas X_n that should be involved in the reading requests is computed. All the following read requests will be performed with Consistency level X_n. In the next section we explain in detail how we estimate the stale reads rate and how we compute the necessary number of replicas.

4.3 Stale Reads Rate Estimation

In this section, we propose an estimation of the stale read rate in the system by means of probabilistic computations. This estimation model requires basic knowledge of the application access pattern and of the storage system network latency. Network latency in this case is of high importance, since it is the determinant of the updates propagation time to other replicas. The access pattern, which includes read rates and write rates is a key factor to determine consistency requirements in the storage system. For instance, it is obvious that a heavy read-write access pattern would produce higher stale reads when adopting eventual consistency.

4.3.1 Stale read probability

We define the situation that leads to a stale read in Figure 4.1. The read may be stale if its starting time X_r is in the time interval between the starting time of the last write and the end of the propagation time of data to the other replicas. This situation is repeatable for any of the writes that may occur in the system. T_p in Figure 4.1 is the time necessary for the propagation of a write or an update to all the replicas. It is computed based on the network latency L_n and the average write size avg_w and should be represented as $T_p(L_n, avg_w)$, but in order to simplify the representation, it will be denoted as T_p in the rest of the chapter.

Transaction arrivals are generally considered as a Poisson process as it is the common way to model them in literature [126, 118]. We assume that the write and the read arrivals follow the Poisson distribution of parameter λ_w^{-1} (we chose λ_w^{-1} instead of λ_w in order to simplify subsequent formulas where the parameter will be inverted) and λ_r respectively. These

parameter values change dynamically at runtime following the read and write requests arrivals monitored in the storage system. Since the distribution of waiting time between two Poisson arrivals is an exponential process, the stochastic variables X_w and X_r of a write time and read time follow an exponential distribution of parameters λ_w^{-1} and λ_r respectively. The probability of the next read being stale corresponding to the aforementioned situation is given by Formula (4.1) with N being the replication factor in the system and X being the number of replicas involved in the read operation. Here X_n =1 for the basic eventual consistency. The left part of Formula (4.1) corresponds to the situation where data is read from replicas for which data is being or going to be propagated while the right part corresponds to the situation where data is being read from the replica that is currently handling the write.

$$\begin{aligned} Pr(stale\_read) = \sum_{i=0}^{\infty}(\frac{N-(X_n=1)}{N} \times Pr(X_w^i < X_r \\ < X_w^i + T + T_p) + \frac{X_n=1}{N} \times Pr(X_w^i < X_r < X_w^i + T)) \end{aligned} \tag{4.1}$$

Having all the writes times (that may occur in the system) following the exponential distribution, the sum of X_w^i all the writes follows a *Gamma* distribution of parameters i and λ_w. Hence, the probability in Formula (4.1) becomes:

$$\begin{aligned} Pr(stale\_read) = \sum_{i=0}^{\infty}(\frac{N-1}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T+T_p) \\ -F_r(t))dt + \frac{1}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T) - F_r(t))dt) \end{aligned} \tag{4.2}$$

The time T to write in the local memory is negligible in comparison to T_P and therefore, we can consider it as equal to 0. A simple replacement of the probability mass function (pmf) of Poisson distribution and the cumulative distribution function (cdf) of *Gamma* distribution results in the following probability:

$$Pr(stale\_read) = \sum_{i=0}^{\infty}\frac{N-1}{N}\int_0^{\infty} t^{i-1}\frac{e^{-\frac{t}{\lambda_w}}}{\gamma(i)\lambda_w^i}(e^{-\lambda_r t} - e^{-\lambda_r(t+T_p)})dt \tag{4.3}$$

After simplifying Formula (4.3), it becomes:

$$Pr(stale\_read) = \sum_{i=0}^{\infty}\frac{(N-1)(1-e^{-\lambda_r T_P})}{N(1+\lambda_r\lambda_w)^i}\int_0^{\infty} t^{i-1}\frac{e^{\frac{1+\lambda_r\lambda_w}{\lambda_w}t}}{\gamma(i)(\frac{\lambda_w}{1+\lambda_r\lambda_w})^i}dt \tag{4.4}$$

The right part of the function in (4.4) is the the *cumulative distribution function* of a *Gamma* law of parameters $\frac{1+\lambda_r\lambda_w}{\lambda_w}$ and i, its value is equal to 1. Moreover, if we consider that:

$$\sum_{i=0}^{\infty}(\frac{1}{1+\lambda_r\lambda_w})^i = \frac{1}{\lambda_r\lambda_w} + 1 \tag{4.5}$$

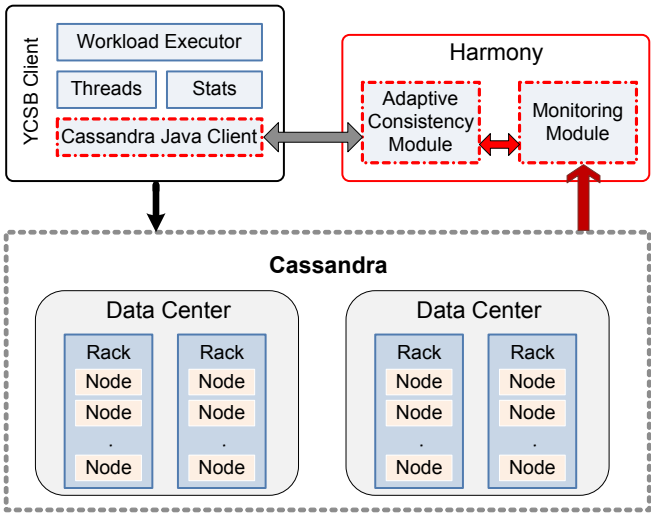


Figure 4.2: *Harmony* implementation and integration with Cassandra and Yahoo! Cloud Serving Benchmark

The final value of the probability for next read being stale, after simplification, is given by:

$$Pr(stale\_read) = \frac{(N-1)(1-e^{-\lambda_r T_p})(1+\lambda_r\lambda_w)}{N\lambda_r\lambda_w} \tag{4.6}$$

4.3.2 Computation of the number of replicas X_n

To compute the number of replicas to be involved in a read operation necessary to maintain the desired consistency, we compute X_n in Formula (4.1) to maintain the inequality (4.7) in order to provide a stale read rate smaller or equal to the *app_stale_rate* denoted as ASR for simplicity.

$$\begin{aligned} Pr(stale\_read) = \sum_{i=0}^{\infty}\Big(\frac{N-X}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T+T_p) \\ -F_r(t))dt + \frac{X}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T)-F_r(t))dt\Big) \leq ASR \end{aligned} \tag{4.7}$$

After simplification, and following similar steps for computing in Formulas (4.3), (4.4), and (4.6), the number of the replicas X_n is given by the Formula:

$$X_n \geq \frac{N((1-e^{-\lambda_r T_P})(1+\lambda_r\lambda_w) - ASR\lambda_r\lambda_w)}{(1-e^{-\lambda_r T_P})(1+\lambda_r\lambda_w)} \tag{4.8}$$

4.4 Implementation & Experimental Evaluation

4.4.1 Harmony Implementation

Harmony can be applied to different cloud storage systems that featured with flexible consistency rules. Currently we have built *Harmony* in Apache Cassandra "Cassandra-1.0.2" [18]. As described in Chapter 3, Cassandra gives the user flexible usage of consistency levels in a per-operation manner. In addition, Cassandra is proven to be very scalable, offering very good performance, and being widely used with large–scale applications such as Facebook and Twitter. Figure 4.2 gives an overview of the *Harmony* implementation. *Harmony* is introduced as an extra layer on Cassandra that aims to provide the most appropriate level of consistency for reading data. The core of this layer consists of two modules. Both modules were implemented in Python 2.7.

The monitoring module collects relevant metrics needed for *Harmony*. The Cassandra *Nodetool* [100] was used to collect the number of reads and writes in Cassandra storage, and the Ping tool was used to collect network latencies in the storage system network. The monitoring module was designed in a multithreaded manner in order to make it time-efficient and to reduce the monitoring time. Each thread collects data from a set of nodes and at the end an aggregation process is applied. The monitoring time is measured and taken into account when computing the read rates and write rates. This data is further communicated to the **adaptive consistency module**. This module is the heart of *Harmony* implementation. An estimation of the stale read rate is computed and then compared to the application stale read that can be tolerated (app_stale_rate) in order to provide an adequate consistency level for the running application at that point of time.

4.4.2 Harmony Evaluation

Evaluation Methodology

We consider two complementary approaches to provide storage as a service for cloud clients. In our first approach, cloud clients which can be applications running on Cloud Computing service such as Amazon EC2 [9] or Google App Engine [67], can connect to the storage service on an S3-like interface to lease their storage resources. This is a typical interface to a highly distributed scalable storage backend that will physically host and manage data. We set up a cloud storage testbed on the Grid'5000 experimental grid and cloud testbed [79] that federates 10 sites in France. In our second approach, the storage service is provided within the virtual disks attached to the virtual machines (VMs) side by side with cloud clients. We set up the underlying storage system on Amazon EC2 clusters and serve applications running inside VMs.

Micro Benchmark. As a benchmark representing typical workloads in current services hosted in clouds, based on several case studies [44, 43], we have selected the Yahoo! Cloud Serving Benchmark (YCSB) framework [137]. YCSB is used to benchmark Yahoo!'s cloud storage system PNUTS [44]. It is extended to be used with a variety of *open-source* data stores such as mongoDB [96], Hadoop HBase [21] and Cassandra [85]. YCSB provides the characteristics of a real cloud–serving environment such as scale-out, elasticity and high availability. For this purpose, several workloads have already been proposed in order to apply a heavy

read load, heavy update load, and read latest load, among other workloads. Moreover, the benchmark is designed to make the integration of new workloads very easy. We use YCSB-0.1.3, as shown in Figure 4.2. We have modified the provided Java client for Cassandra in order to allow read operations to be performed with different consistency levels at run time. The Java client uses Thrift [23] to communicate with the cluster and is provided with the set of hosts from which it should request a read or a write. The modified Java client reads data from Cassandra with the consistency level provided dynamically by the adaptive consistency module.

Experimental Setup

We have evaluated *Harmony* with Cassandra deployed on both Grid'5000 and Amazon EC2.

Setup on Grid'5000. We use two clusters in the *Sophia* site with a total of 84 nodes and 496 cores. All nodes are equipped with x86_64 CPUs and 4 GB of memory. The nodes are interconnected with Gigabit Ethernet. All nodes from the first cluster have two hard disks with combined capacity of 600 GB per node. As for the second cluster, the nodes are all equipped with hard disks of 250 GB.

Setup on Amazon EC2. We have used 20 Virtual machines of *m1.large* type located in the us-east-1a availability zone in the east cost. Each virtual machine has 2 cores and 7.5 GB of memory. The total size of disk storage available is 14.78 TB.

In both experiments, Cassandra was configured in order to have a replication factor of 5. Moreover, *OldNetworkTopologyStrategy* was chosen as a replication strategy. This strategy ensures that data is replicated over all the clusters and racks. We have deployed Cassandra on the two clusters on the Grid'5000 and on the 20 nodes of Amazon EC2. We have used YCSB with workload-A which provides a heavy read-update data access. For the experiments conducted on Grid'5000, we have initially inserted a load of 3 million rows and a total size of 14.3 GB after replication. Each workload run had a total of 3 million operations consisting of reads and updates. For the experiments that were conducted on Amazon EC2, an initial load of 5 million rows was inserted, with a total size of 23.85GB after replication. Each workload that was run consisted of 10 million operations.

Stale Reads Estimation in *Harmony*

We have first studied the impact of the workload access patterns, the number of clients, and network latency on the stale reads estimation. Accordingly, we use two workloads: workload-A, which has a heavy read-update access pattern, and workload-B, which has a heavy read access pattern with a small portion of writes representing approximately 5% of the total number of operations. We ran both workloads, varying the number of threads starting with 90 threads, then, 70, 40, 15 and finally, one thread.

The impact of workload access pattern and client number. As shown in Figure 4.3, the probability of reading stale data for workload-B is relatively smaller than the one for workload-A. This is because the number of updates is smaller. We observe that the number of updates plays very important role in causing stale reads even with a high number of reads. Moreover, we observe that the probability of reading stale data varies according to the number of threads. We can see that for workload-A, the probability of stale reads gradually decreases with the

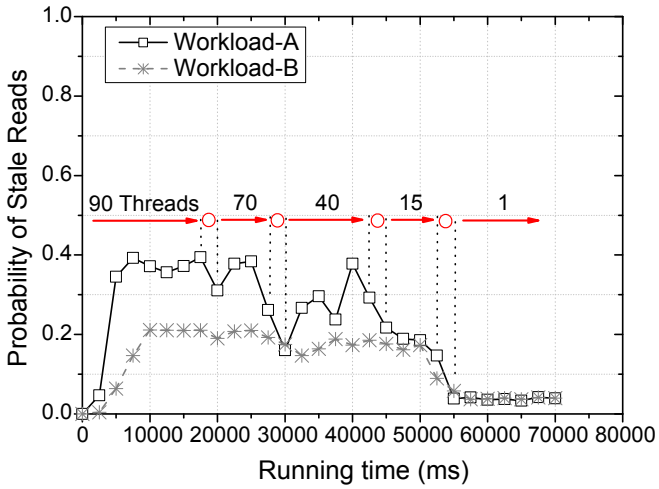


Figure 4.3: Workloads and Number of Clients impact on stale read probability

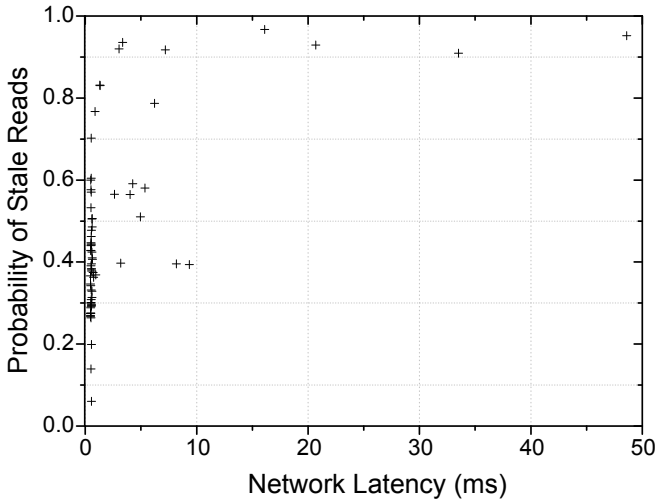


Figure 4.4: Network latency impact on stale read probability

number of threads, because increasing the thread number increases the throughput and thus increases the reads and writes rate. Also, we notice the probability reduction gap is big during the transition (changing the number of threads).

The impact of network latency. In order to see the impact of network latency on the stale reads estimation we ran workload-A –varying the number of threads starting with 90 threads, then, 70, 40, 15 and finally, one thread– on Amazon EC2 and measure the network latency during the run-time. Figure 4.4 presents the results. We can see that high network latency causes higher stale reads regardless of the number of the threads (higher latency dominates the probability of stale reads), while when the latency is small the probability will be varied according to the reads and writes rates (with smaller impacts of the network latency).

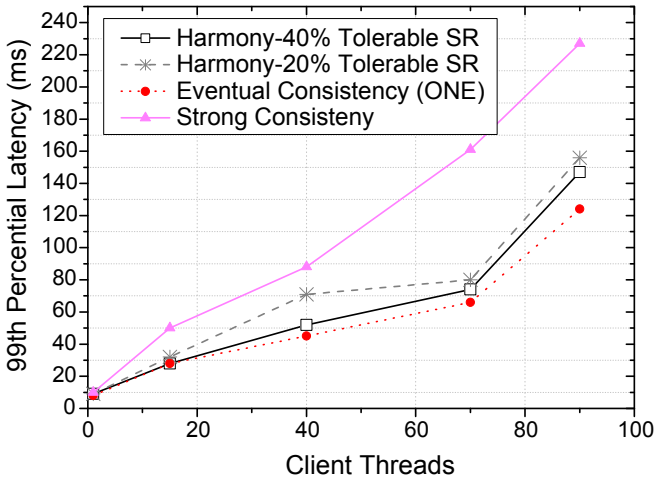


Figure 4.5: Read operation latency on Grid'5000

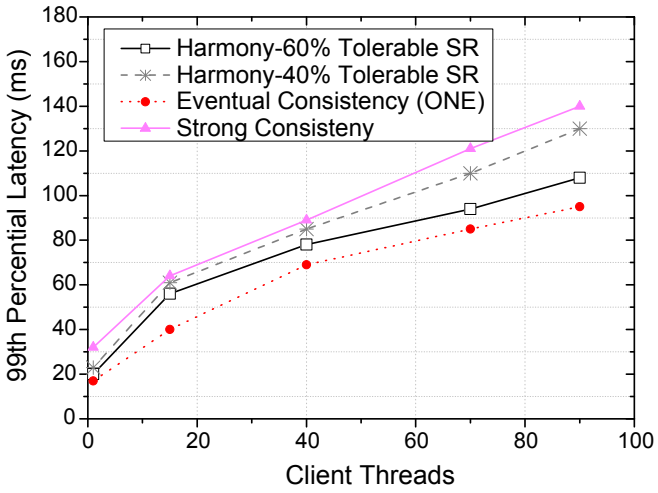


Figure 4.6: Read operation latency on Amazon EC2

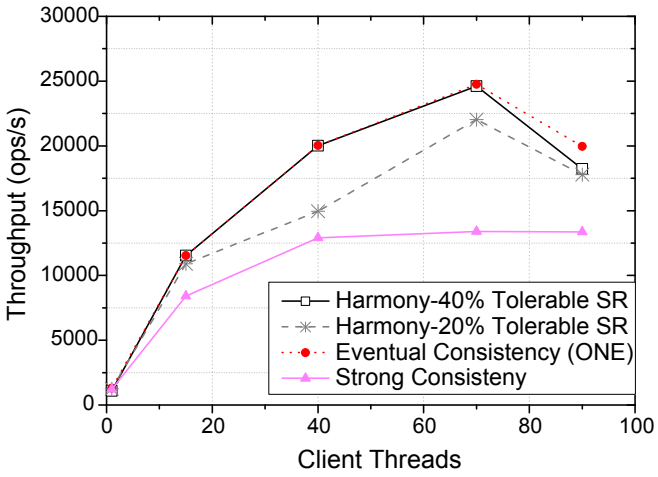


Figure 4.7: Throughput on Grid'5000

Latency and throughput of the application in *Harmony*

As mentioned earlier, in *Harmony*, the application requirements are defined as the stale reads rate that an application can tolerate during its running. Accordingly, we compare *Harmony* with two settings (two different tolerable stale read rates) with strong and eventual consistency on our both storage approaches (Grid'5000 and Amazon EC2). The first tolerable stale read rates are 40% for Grid'5000 and 60% for Amazon EC2 (these rates tolerate more staleness in the system implying lower consistency levels and thus less waiting time), and the second tolerable stale read rates are 20% for Grid'5000 and 40% for Amazon EC2 (these rates are more restrictive than the first ones, meaning that the number of read operations performed with a higher level of consistency is larger). Network latency is higher in Amazon EC2 than in Grid'5000 (5 times higher in the normal case), thus we choose higher stale read rate for the same workload with Amazon EC2. We run workload-A while varying the number of client threads.

Figures 4.5 and 4.6 present the 99th percentile latency of read operations when the number of client threads increases on Grid'5000 and EC2 respectively. While the strong consistency approach provides the highest latency having all the reads to wait for the replies from all the replicas spread over different racks, the eventual consistency approach is the one that provides smaller latency because all the read operations are performed on one close replica but at the cost of consistency violation. We can clearly see that *Harmony* with both settings provides almost the same latency as the eventual consistency. Moreover, the latency increases by decreasing the tolerable stale reads rate of an application as the probability of stale read can easily get higher than these rates, which requires a higher consistency levels and, as a result, a higher latency.

In Figures 4.7 and 4.8, we show the overall throughput for read and write operations with different numbers of client threads. The throughput increases as the number of threads increases. However, the throughput decreases with more than 90 threads. This is because the number of client threads is higher than the number of storage hosts and threads are served concurrently. We can observe that the throughput is smaller with strong consistency. The fact that read operations with higher consistency levels have high latencies, makes the number

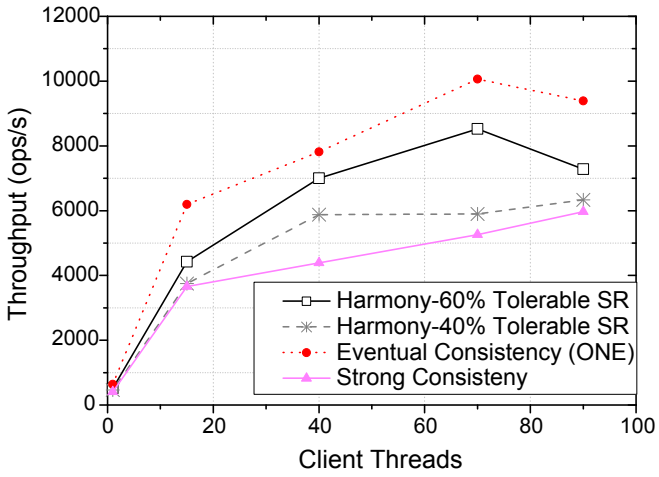


Figure 4.8: Throughput on Amazon EC2

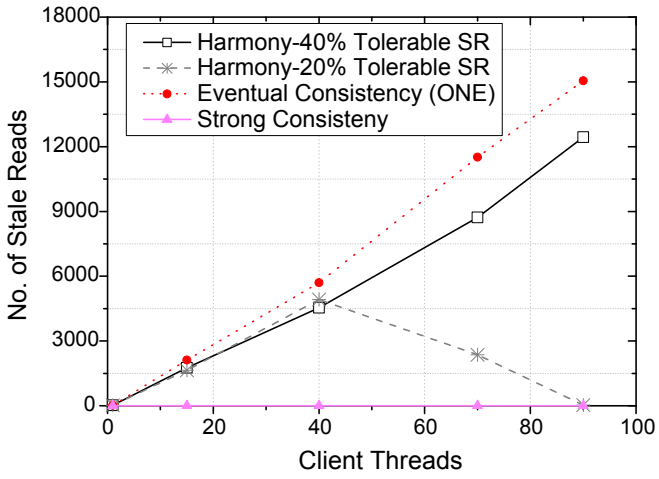


Figure 4.9: Observed staleness on Grid'5000

of possible operations per second smaller. We can notice that our approach with a stale reads rate of 40% and 60% for Grid'5000 and Amazon EC2 respectively, provides very good throughput that can be compared to the one of static eventual consistency approach. But, while exhibiting very good throughputs, our adaptive policies provide a better consistency and fewer stale reads due to the fact that higher consistency levels are chosen only when it matters.

Actual Staleness in *Harmony*

In Figures 4.9 and 4.10 , we show that *Harmony*, with all the policies with different application tolerated stale reads rates, provides less stale reads than the eventual consistency approach. Moreover, we can see that, with a more restrictive tolerated stale reads rate, we get a smaller number of stale reads. We observe that with rates of 20% and 40% for Grid'5000 and Amazon EC2 respectively, the number of stale reads decreases when the number of threads grows over 40 threads. This is explained by the fact that with more than 40 threads the es-

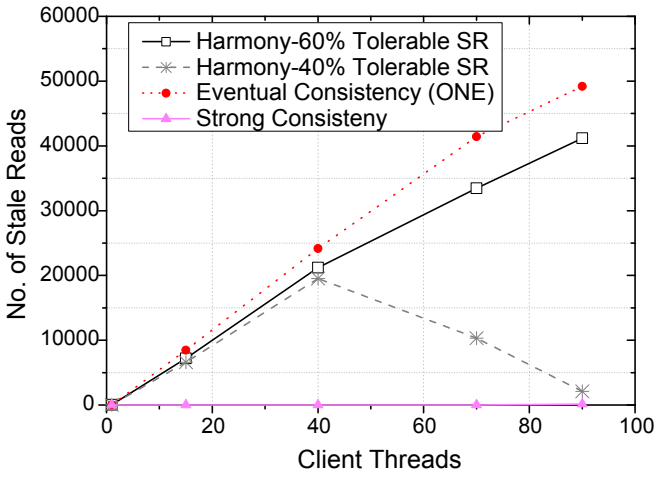


Figure 4.10: Observed staleness on Amazon EC2

timated rate of stale reads gets higher than 20% and 40% respectively, for most of the run time, and higher consistency levels are chosen, thus decreasing the number of stale reads. It needs to be pointed out that this number of stale reads is not the actual number of stale reads in the system in the normal run, but it is representative.

In fact, to measure the number of stale reads within YCSB, we perform two read operations for every read operation in the workload. The first read is performed with the relevant consistency level chosen by our approach, and the second read is performed with the strongest consistency level. Then, we compare the returned timestamps from both reads, and if they do not match, it means that the read is stale. Although this helps to estimate the number of stale reads, it completely changes the latency of reads and the throughput in the system. Moreover, it directly affects the monitoring data about system state. Additionally, the second read with strong consistency level provides more time for the next write to be propagated to the other replicas and, thus more chances for the next read to be fresh.

4.4.3 Estimation Accuracy of Stale Reads Rate

In order to measure the accuracy of stale reads rate, we designed and implemented our own workload where we can have a total control and store and access information that help us detect stale reads. The workload consists in multiple threads (where every thread instantiates a Cassandra client). Each thread is configured to perform 100 operations with read/write ratio probability of 60/40. The waiting time between operations is generated randomly following an exponential distribution of average waiting time of 1/10 (ms). We started Cassandra on 5 nodes in the *Sophia* site and we used 'OldNetworkTopologyStrategy' as a replication strategy with a replication factor of 5. The system was initialized with 5 keys.

We run our workload with an increasing number of clients (threads), starting with 5, then 10, 20, 50, and finally 100 clients. Stale reads are detected based on timestamp comparisons in the log files. For every read, its returned timestamp is compared to the timestamp of the last write (update) to that specific key. If the two timestamps do not match, the read is flagged as stale.

Figure 4.11 shows both the observed stale rate and our estimation with the increasing

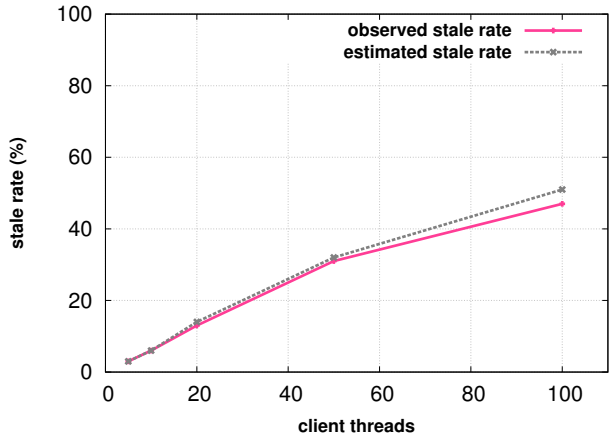


Figure 4.11: Stale read rate estimation accuracy

number of clients. We can observe that our estimation model provides close values to the observed ones, which demonstrates the effectiveness of our probabilistic computations. In addition, the accuracy of the estimation is higher when the number of clients is smaller. When the number of clients grows higher, the random generation of waiting times between operations in different clients in the workload makes standard deviation from the mean arrival time rate higher, which increases the margin of error. However, such error remains small and does not affect the overall estimation as can be seen in Figure 4.11.

4.5 Discussion

Eventual consistency was employed in cloud storage system as an alternative to traditional strong consistency to achieve scalable and high-performance services [124]. Many commercial cloud storage systems have already adopted the eventual consistency approach such as Dynamo [49] in Amazon S3 [13], Cassandra [85] in Facebook [59] and PNUTS [44] in Yahoo!. A fair number of studies have been dedicated to measuring the actual provided consistency in cloud storage platforms [126, 17, 29]. Wada *et al.* [126] investigate the consistency properties provided by commercial storage systems and report on how and under what circumstances consumer may encounter stale data. Also, they explore the performance gain of using weaker consistency constraints. Anderson *et al.* [17] propose an offline consistency verification algorithm and test three kind of consistency semantics on registers including safety, regularity, and atomicity in the Pahoehoe key-value store using a benchmark similar to YCSB [43]. They observed that consistency violations increase with the contention of accesses to the same key.

Moreover, in order to meet the consistency requirements of applications and reduce the consistency violation, some studies are done on adaptive consistency tuning in cloud storage systems [82, 111, 128]. Kraska *et al.* [82] propose a flexible consistency management that is able to adapt the resulting consistency level to the requirements stated by applications. The inconsistencies considered in their work are due to update conflicts. Accordingly, they build a theoretical model to compute the probability of update conflict, and then compare it

to a threshold. As a result, they choose either serializability using strong consistency or session consistency, which is a weaker consistency. However, their approach cannot be applied with eventual consistency as weaker consistency. This is due to the fact that in eventual consistency, the staleness is due to the update propagation latency rather than just the conflict of two or more updates on different replicas. Moreover, the threshold –used to determine the type of consistency– is computed based on the financial cost of pending update queues and not related to the storage backend itself. Wang *et al.*[128] propose an application-based adaptive mechanism of replica consistency. This mechanism was proposed with a specific replication architecture. The architecture relies on multi-primary replicas and secondary replicas where the latter are read-only replicas. Consistency is either strong or eventual and the choice between the two is made by comparing the read rate and the write rate to a threshold. The main limitation of this work is the arbitrary choice of a static threshold. In addition, this approach was proposed for their specific proposed replication architecture, which is not commonly used in current cloud storage solutions.

In contrast to the aforementioned work, *Harmony* is using the stale reads rate to define the consistency requirements of the application. Moreover, it dynamically alters the replicas number involved in an operation according to the estimated stale reads rate and the network latency, during run-time. Thus *Harmony* achieves adequate tradeoffs between consistency and both performance and availability.

4.6 Summary

With the explosion of cloud storage businesses and the increasing number of web services migrating to the cloud, a strong consistency model becomes very costly when scalability and availability are required. Thus, weaker consistency models have been proposed, but these models may lead to far too much inconsistency in the system. In this chapter, we presented *Harmony*, a novel approach that handles data consistency in cloud storage adaptively by choosing the most appropriate consistency level dynamically at run time. In *Harmony*, we collect relevant information about the storage system in order to estimate the stale read rate when consistency is eventual, and make a decision accordingly. In order to be application-adaptive, *Harmony* takes into account the application's needs expressed by the stale read rate that can be tolerated. We show that our approach provides better performance than traditional approaches that are based on strong consistency. Moreover, it provides more adequate consistency than static eventual consistency approaches. In addition, our solution is designed to be completely tunable to provide the system or the application administrator with the possibility of controlling the degree of compromise between performance and consistency.

Chapter 5

Consistency vs. Cost: Cost-Aware Consistency Tuning in the Cloud

Contents

This chapter is mainly extracted from the paper: *Consistency in the Cloud: When Money Does Matter!* Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, Maria S. Pérez. In the proceeding of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID 2013), Delft, Netherlands, May 2013, pp.352-359

In the previous chapter we addressed the important issue of the tradeoff between consistency and performance. This tradeoff has long been the center of studies on consistency over the years. However, in the era of Big Data and Cloud Computing, another tradeoff between consistency and monetary cost is equally important. In this chapter, we address this particular topic, thereby, providing a thorough study of the impact of consistency management on the monetary cost. In addition, we leverage this study to explore how to provide cost–efficient management of consistency with significant money savings.

5.1 Motivation

A particularly challenging issue that arises in the context of storage systems with geographically-distributed data replication is how to ensure a consistent state of all the replicas as seen in Chapter 3. Insuring strong consistency by means of synchronous replication introduces an important performance overhead due to the high latencies of networks across data centers (the average round trip latency in Amazon sites varies from 0.3ms in the same site to 380ms in different sites [87]). Consequently, many Internet services tend to rely on storage systems with *eventual consistency*. In this context, consistency–performance and consistency–availability tradeoffs have long been investigated in literature: many consistency optimization solutions have been devoted to improving the application throughput and/or latency while preserving acceptable stale reads rate (including our approach *Harmony* introduced in the previous chapter). However, in the area of Cloud Computing, the economic cost of using the rented resources is very important and should be considered when choosing the consistency policy.

Since Cloud Computing is an economically–driven paradigm where the monetary cost is extremely important, in particular at the scales of Big Data, the monetary cost of the various consistency models should be explored. However, very few studies investigated this aspect. In this chapter, we investigate better approaches in order reduce the monetary cost while preserving acceptable level of consistency. Hereafter, our goals are the following:

Service/bill details. We introduce a detailed study to provide in-depth understanding of the monetary cost of cloud services with respect to their adopted consistency models. We discuss the different resources contributed to a service and the cost of these resources. We introduce an accurate decomposition of the total bill of the service into three parts with respect to the contributed resources: virtual machine (VM) instances cost, storage cost and network cost. To complement our analysis, a series of experiments are conducted to measure the monetary cost of different consistency levels in the Cassandra system [85] —as an illustrative popular cloud storage system with versatile consistency usage— on Grid'5000 [79] and the most popular cloud service Amazon EC2 [9]. Such a study is important as a big-picture understanding of consistency in geo-replicated systems must take into account the monetary cost within the cloud.

Novel metric. We define a new metric that express the relation between consistency and cost and evaluate the efficiency of consistency in the cloud.

Equitable consistency at low cost. Based on the efficiency metric, we introduce a simple yet efficient approach named *Bismar*, which adaptively tunes the consistency level at

runtime in order to reduce the monetary cost while simultaneously maintaining a low fraction of stale reads.

5.2 How Much does Storage Cost in the Cloud ?

5.2.1 Cloud Storage Service and Monetary Cost

Since Cloud Computing is an economically–driven distributed system paradigm, deploying and running services and applications in the cloud comes with a monthly bill. In general, services require a set of linked servers (distributed in multi-sites) to run the web-service applications; these servers are attached to a group of storage devices, which store the services data. With respect to cloud resource offers, a basic service bill includes charges for the following resources[1] :

Computing resources. Virtual machines equipped with a certain amount of CPU and memory resources. Cloud IaaS providers offer different VM instances — varying in the resource's capacity and accordingly the prices — and typically charged for the incurred virtual machine hours. For example, Amazon EC2 [9] offers a set of instances with different configurations and prices: while the cheapest instance (small instance, equivalent to a server with a CPU capacity of a 1.0-1.2GHz and memory size of 1.7GiB) comes at cost of 0.065$ per hour, the most expensive instance (High I/O Quadruple Extra Large Instance, equivalent to a server with CPU capacity of $35 \times 1.0 - 1.2$GHz and memory size of 60.5GiB) comes at cost of 3.100$ per hour.

Storage resources. Cloud IaaS providers offer two types of storage services that are different in their pricing and usability. Taking Amazon Web Services as an illustrating example, there are two representative storage services: Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Block Store (Amazon EBS). The storage services are typically billed according to the used GBs per month and number of requests to the stored data. Taking into account the tremendous amount of data that current services need to manage and maintain, and the need to reduce the latency of data movement when processing data, Amazon EBS becomes the customer's first choice to achieve not only highly scalable and high performance services but highly reliable and predictable ones as well. This is despite the fact that Amazon EBS can be attached to any running Amazon EC2 instance and can be exposed as a device within the instance. Consequently, in this study we adopted the Amazon EBS pricing scheme.

Network resources. Cloud IaaS providers equip their infrastructure with high–speed networks not only within data centers but also across geographically distributed centers. This comes at a monetary cost, although services don't currently reflect the network usage and cost. The network cost is usually embedded within the cost of other services (computational service and storage services), and it varies according to the service type and within/across sites (e.g., the cost of data transfer between Amazon EC2 instances is zero if they are located in the same availability zone).

[1]The pricing of some cloud services (computing and storage services) may vary at different providers or at different provider-sites. As the goal of our study is to explore the consistency cost variation, we assume that the computing and storage pricing is the same at different sites

Monetary cost of consistency: why does it matter?

Strong consistency by the means of synchronous replications may introduce high latencies due to the cross-sites communication and therefore will significantly increase the monetary cost of the services:

- High latency causes high monetary cost. This is due to the fact that the cost of leasing a VM-instance is proportional to the latency, which in turn affects the throughput of the system resulting in high runtime, in addition to the increased cost of both the storage (*e.g.* number of requests to the copies) and the communication cost (*e.g.* number of cross-sites communication) due to the synchronous cross-site replication.
- High latency causes significant financial losses for service providers that use such storage systems. For instance, the cost of a single hour of downtime for a system doing credit card sales authorizations has been estimated to be between 2.2M$-3.1M$ [106].

On the other hand, we observe that eventual consistency or weaker consistency may reduce the monetary cost with respect to a lower maintained latency and therefore lower instance costs, but this comes at the risk of increasing the rate of stale data (*e.g.*, [126] demonstrated that under heavy reads and writes some of these systems may return up to 66.61% stale reads). This in turn, adversarily impacts the financial profit of the service providers: it generates significant financial losses as it violates the SLAs of services users. This makes eventual consistency a two-edged sword. While the eventual consistency has been exploited extensively in literature and commercial products, its monetary cost and negative impacts on the stale reads rate have been largely ignored.

The *aforementioned observations*, combined with the urgent need to address the consistency-cost efficiency and stale reads problems associated with quorum replications, motivate us to an in-depth study of the monetary cost of the different consistency levels in the cloud and — as a result — to propose our cost efficient optimization.

5.2.2 Cost Model

After building a big-picture understanding of the cost of services deployed in the cloud by describing the different resources contributed to obtain a certain level of consistency in geo-replicated storage systems. Hereafter, we complement our macroscopic analysis with a detailed analysis of the consistency cost in the cloud, using a widely used open source geo-replicated storage system that supports multi-level consistency as an illustrated example, namely Cassandra [4].

Ideally, we would like to get a deep idea of why different consistency levels may result in different costs, how the resources accordingly contribute to the total cost, and how background operations such as read repair can impact the overall cost.

The choice of consistency level (*cl*) affects all of these three costs. When higher consistency levels are required more replicas are involved in the requests. That affects both operations latency and throughput, which leads to a higher runtime. Similarly, network traffic grows higher with higher consistency levels, which leads to a higher networking bill. Moreover, higher consistency levels generate a higher number of requests from storage devices, directly affecting storage cost.

Formula 5.1 presents the overall cost for geo-replicated based services for a given consistency level cl. Essentially, this cost is the combination of the VM instances cost $Cost_{in}(cl)$, the backend storage cost $Cost_{st}(cl)$, and network cost $Cost_{tr}(cl)$.

$$\text{Cost}_{all}(\text{cl}) = \text{Cost}_{in}(\text{cl}) + \text{Cost}_{tr}(\text{cl}) + \text{Cost}_{st}(\text{cl}) \tag{5.1}$$

5.2.2.1 Computing unit: instances cost

A common pricing scheme used by recent cloud providers is primarily based on virtual machine (VM) hours. Formula 5.2 presents the cost of leasing *nbInstances* VM-instances for a certain time (*runtime*).

$$\text{Cost}_{in}(\text{cl}) = \text{nbInstances} \times \text{price} \times \lceil \frac{\text{runtime}}{\text{timeUnit}} \rceil \tag{5.2}$$

Here the *price* is the dollar cost per *timeUnit*[2] (e.g., In Amazon EC2 small instance the price is 0.065 per *hour*).

In order to generalize our pricing model and avoid inaccurate pricing due to unexpected network behavior (especially that we are studying the consistency cost in geo-distributed sites), we present the *runtime* in the form of number of operations *nbOps* in the workload while fixing the *throughput* of a specific consistency level.

$$\text{runtime} = \frac{\text{nbOps}}{\text{throughput}} \tag{5.3}$$

The throughput varies from one consistency level to another according to the size of the internal traffic between sites.

5.2.2.2 Storage cost

As mentioned earlier the storage cost includes the cost of leased storage volume (GB per month) and the cost of I/O requests to/from this attached storage volume. In Amazon EC2 for instance, this would be the cost of attaching Amazon EBS to VM-instances in order to increase the storage capacity using a highly durable and reliable way. The total storage cost is accordingly given by Formula 5.4 :

$$\text{Cost}_{st}(\text{cl}) = \text{costPhysicalHosting} + \text{costIORequests} \tag{5.4}$$

Based on the size of hosted data (including all replicated data) $nbNodes \times dataSize$ where $dataSize$ is the average data size per volume attached to VM-instance (locality and load balancing are important features in current data centers), we calculate the $costPhysicalHosting$ in Formula 5.5.

$$\text{costPhysicalHosting} = \text{nbNodes} \times \lceil \frac{\text{dataSize}}{\text{sizeUnit}} \rceil \times \text{price} \tag{5.5}$$

[2]We use the ceiling function because most providers charge each partial instance-hour as a full hour.

where the *price* is the dollar cost per *sizeUnit* (*e.g.* in Amazon EBS the price is 0.10 per $GB - month$).

We further estimate $costIORequests$ in Formula 5.6.

$$\text{costIORequests} = \frac{\text{cl} \times \text{nbOps} + \text{readRepairIO}}{\text{nbRequestsUnit}} \times \text{price} \tag{5.6}$$

where $nbOps$ is the number of operation with respect to the consistency level cl (it varies according to the number of replicas involved in an operation). The read repair in a background operation is mostly triggered when inconsistency is detected. It generates requests to the storage devices and therefore it is important to include the read repair operations in our formula readRepairIO (more details on the read repair function will be provided further in this section).

5.2.2.3 Network cost

The network cost varies in accordance to the service type of the source and destination (e.g., computational service and storage services) and whether the data transfer is within or across sites. In general, inter–datacenter communications are more expensive than intra–datacenter communications. Formula 5.7 shows the total cost of network communications as the sum of inter– and intra–datacenter communications [3]($traffic InterDC$ and $traffic IntraDC$).

$$\text{Cost}_{tr}\ (\text{cl}) = \text{price(interDC)} \times \lceil \frac{\text{trafficInterDC}}{\text{sizeUnit}} \rceil + \text{price(intraDC)} \times \lceil \frac{\text{trafficIntraDC}}{\text{sizeUnit}} \rceil \tag{5.7}$$

where $price(interDC)$ and $price(intraDC)$ are the dollar cost per $sizeUnit$.

Hereafter we illustrate how to estimate both the inter- and intra-datacenter traffic.

Formula 5.8 shows our model of the inter–datacenter, $traffic InterDC$, given the replicas communication $interDcRep$, the request routing $requestrouting$, and the internal mechanisms traffic $IMechTraffic$.

$$\text{trafficInterDC} = \text{interDcRep} + \text{requestRouting} + \text{IMechTraffic} \tag{5.8}$$

The inter-site traffic generated by the replicas communications strongly depends on the consistency level and the distribution of replication among data centers (i.e., the number of replicas involved in a request to other data centers which can be estimated as $\lfloor (nbDc - 1) \times \frac{cl}{nbDc} \rfloor$[4] where $nbDc$ is the number of data centers). Formula 5.9 shows our estimation of the inter traffic generated by the replicas communications.

$$\text{InterDcRep} = \lfloor (\text{nbDc - 1}) \times \frac{\text{cl}}{\text{nbDc}} \rfloor \times \text{AvgDataSize} \times \text{nbOps} \tag{5.9}$$

[3] For simplicity, we consider only two geographical areas within which the prices differ. Some cloud providers may have more geographically- oriented prices: within availably zone, within regions, between regions. However, our pricing model can be easily extended to any number of geographical-oriented pricing options.

[4] For example if the ($nbDC = 3$) and number of replicas involved in an operation ($cl = 4$), the estimated number of replicas involved in a request on other data centers is $\lfloor 2 \times \frac{4}{3} \rfloor = \lfloor \frac{8}{3} \rfloor = 2$ where $\lfloor\ \rfloor$ is a floor function.

where *avgDataSize* is the average data size needed to be propagated to other replicas for one operation.

The traffic generated by the request routing and internal mechanisms depends essentially on the storage system design and implementation. Since our approach is destined to run on Cassandra storage, hereafter we illustrate such values with respect to this particular storage system. In Cassandra, all nodes (peers) have equal ranges of data and thus have an equal number of keys: this implies that each node is responsible for $\frac{1}{number\ of\ nodes}$ fraction of the keys.

Giving the number of nodes as *nbNodes* and the average number of nodes per datacenter *avgNodesDc*, the average number of request routing for an operation can be estimated as $\frac{nbNodes - avgNodesDc}{nbNodes}$. The size of inter traffic generated by request routing for a number of operations *nbOps* is therefore denoted as Formula 5.10.

$$\text{requestRouting(interDC)} = \frac{\text{nbNodes} - \text{avgNodesDc}}{\text{nbNodes}} \times \text{nbOps} \times \text{avgDataSize} \tag{5.10}$$

In Cassandra storage, the main internal traffic is generated by the gossip traffic and read repair mechanism as shown in Formula 5.11. The gossip traffic — used to share the state of nodes in the ring — is relatively small since it is just transmitting the state of one node, which is negligible compared to data transfer.

$$\text{IMechTraffic} = \text{gossip(interDc)} + \text{readRepair(interDc)} \tag{5.11}$$

On the other hand, the read repair is used to propagate data to out of date (stale) replicas. The read repair function is triggered in two cases:

1. At random times for some requests: defined by the system administrator.
2. Whenever inconsistency is detected.

Formula 5.12 shows that read repair traffic depends on the probability or chance of triggering the mechanism *rrChance* which is defined by the storage administrator, as well as the chance of detecting mismatching replica timestamps $mmChance = \frac{rf - cl}{rf} \times \frac{nbWrites}{nbReads + nbWrites}$, where rf is the replication factor, *nbWrites* and *nbReads* are the number of write and reads.

$$\begin{aligned} \text{readRepair(interDC)} &= \text{nbOps} \times \text{avgDataSize} \\ \times (\text{rrChance} \times \lfloor \frac{\text{rf}}{\text{nbDc}} \rfloor &+ \text{mmChance} \times \lfloor \frac{\text{rf} - \text{cl}}{\text{nbDc}} \rfloor) \end{aligned} \tag{5.12}$$

Computing the intra–datacenter traffic size is very similar to the one of inter–datacenter traffic. However, the intra traffic size of request routing is given by Formula 5.13.

$$\text{requestRouting(intraDC)} = \frac{\text{avgNodesDc} - 1}{\text{nbNodes}} \times \text{nbOps} \times \text{avgDataSize} \tag{5.13}$$

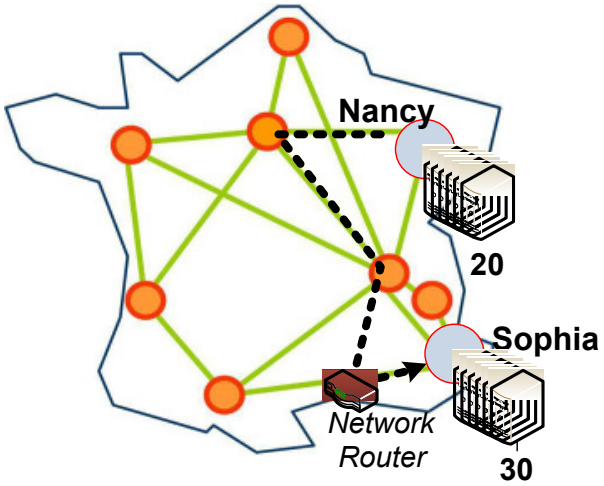


Figure 5.1: Experiments setup on Grid5000

Similarly, we only consider the traffic in-between replicas within the same datacenter: Accordingly, the intra-site traffic generated by the replicas communications is denoted as in Formula 5.14.

$$\text{intraDcRep} = (\lceil \frac{\text{cl}}{\text{nbDc}} \rceil - 1) \times \text{avgDataSize} \times \text{nbOps} \tag{5.14}$$

The read repair traffic is given by Formula 5.15.

$$\begin{aligned} \text{readRepair(intraDC)} = \text{nbOps} \times \text{avgDataSize} \times (\text{rrChance} \times (\text{rf} - \lfloor \frac{\text{rf}}{\text{nbDc}} \rfloor) \\ + \frac{\text{rf} - \text{cl}}{\text{rf}} \times ((\text{rf} - \text{cl}) - \lfloor \frac{\text{rf} - \text{cl}}{\text{nbDc}} \rfloor)) \end{aligned} \tag{5.15}$$

5.2.3 Consistency vs. Cost: Practical View

As we mentioned, our goal is to investigate the monetary cost variation of geo-replicated storage systems when adopting different consistency levels. We therefore complement our earlier analysis, by evaluating the monetary cost in Cassandra.

Experimental setup

We run our experiments on Grid'5000 [79] and Amazon Elastic Compute Cloud (EC2). On Grid'5000, we deployed Cassandra on two data centers (sites): with 30 nodes on the *Sophia* site and 20 nodes on the *Nancy* site as shown in Figure 5.2.3. All the nodes in *Sophia* are equipped with a 250 GB hard disk, 4 GB of Memory, and 4-cores AMD Opteron. The nodes in Nancy are equipped with disks of 320 GB space, 16 GB of Memory, and 8-cores Intel Xeon. The network connection between the two sites is provided by RENATER (The French national telecommunication network for technology, education, and research). It consists of a standard architecture of 10 Gbit/s dark fibers. The network route between the two sites is

Table 5.1: Pricing schemes used in our evaluation

Computing unit *Large instance*	Storage unit	Storage Requests	Intra comm	Inter Comm
0.32$ per hour	0.10$ per GB/-month	0.10$ per 1 million Requests	0.00$ per GB	0.01$ per GB

the following: *Nancy-Paris-Lyon-Marseille-Sophia*. The average round trip latency is on average 0.230 ms within the same site and 18.2 ms in-between the two sites. On Amazon EC2, we also deployed Cassandra on 18 large instances (the m1.large type) on two availability zones: 10 instances on *us-east-1a* and 8 instances on *us-east-1d*. The average round trip latency is on average 0.284 ms within the same site and 0.813 ms in-between the two availability zones.

We used Cassandra-1.0.2 with a replication factor of 5 replicas: 2 replicas are allocated in *Nancy* and 3 replicas in *Sophia* (The same replication factor is used in Amazon EC2: 2 replicas in *us-east-1d* and 3 replicas in *us-east-1a*). Our replication strategy uses *NetworkTopologyStrategy* to enforce replication across multiple data centers. We adopt the pricing schemes from Amazon web services as shown in Table 5.1[5]. We study the cost variation by evaluating different consistency levels (e.g., eventual consistency: one, two, Quorum: three, and strong consistency: All).

Micro Benchmark

As for the evaluation of Harmony in the previous chapter, our need for micro benchmark that exhibits typical characteristics of cloud workloads, led us to use Yahoo! Cloud Serving Benchmark (YCSB). YCSB can be used with multiple cloud storage solutions such as mongoDB [96] , Hadoop HBase [21] and Cassandra [85]. In addition, YCSB exhibits real cloud features including as scale-out, elasticity and high availability. In our experimental evaluation, we use YCSB-0.1.3 and we run WorkloadA, which is a heavy read-update workload (read/update ratio: 60/40). In both environments, our workload consists of 10 million operations on 5 million rows with a total of 23.84GB of data after replication.

Results on Grid'5000

As shown in Figure 5.2, the total monetary cost decreases when degrading the consistency level: the cost reduces from $138.76 — when the consistency level is set to *ALL* — to $71.72 when the consistency level is *ONE* (i.e., weak consistency reduces the cost by almost 48%). This result was expected as lower consistency levels involve fewer replicas in the operations, and thus maintaining low latency, less I/O requests to the storage devices, and less network traffic in general (the runtimeof WorkloadA varies from 4 hours to 7 hours according the consistency level). This cost reduction, however, comes at the cost of a significant increase in the stale reads rate: as shown in Figure 5.2 79% of the reads are stale reads — only 21% of the reads are fresh reads — when the consistency level is set to *ONE*.

[5]The price of Amazon EC2 large instance was $0.32 at the time of conducting these experiments and it is now $0.26. However, as this price is applied to all consistency levels the difference in the pricing therefore doesn't affect our results and findings.

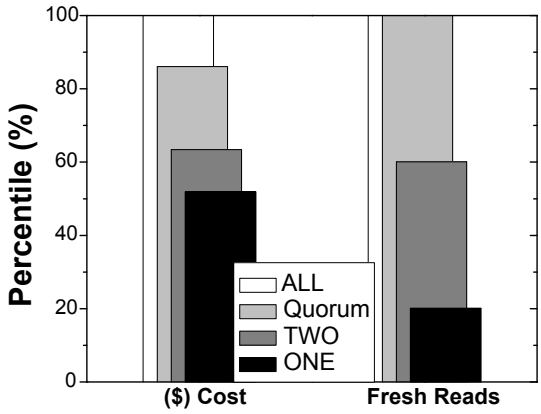


Figure 5.2: Monetary Cost and Fresh reads rate on Grid'5000

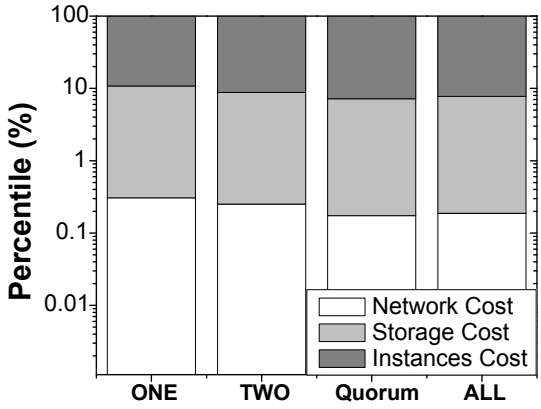


Figure 5.3: Breakdown of the Monetary cost on Grid'5000 (log scale)

Furthermore, it is obvious that degrading the consistency level for Quorum (here the number of replicas involved in an operation is 3 replicas) reduces the total cost by 13% while maintaining a zero stale reads rate as shown in Figure 5.2. This is because the storage system answers the read requests with the most up-to-date replica (fresh reads), which is always in the replicas quorum. Moreover, degrading the consistency level to TWO reduces the total monetary cost by almost 36%, but it adversary impacts the system consistency: only 61% of the reads where fresh reads.

Observation 1 The total cost of geo-replicated services strongly depends on the consistency level adopted: stronger consistency has higher a cost but a higher rate of fresh reads and vice versa. However, as services differ in their tolerable stale reads and their access pattern (within the same service: there is a significant diurnal variation in the access pattern and the load levels), there is a need to define new metric to define the consistency level of an application.

Figure 5.3 shows the breakdown of the total cost according to the contributed resources. In general, the instances cost has the higher cost amongst other resources (storage and network): it contributes to almost 90% of the service bill while the storage and network con-

tribute on average to only 9% and 0.4%, respectively. This is due to our experiments' scale — number of operations — and the cheap prices of resources (as shown in Table 5.1 the intra communication is free of charges).

As shown in Figure 5.3, *storage cost* has a relatively lower contribution to the total cost for stronger consistency (ALL and Quorum) compared to weaker consistency (ONE and TWO): it contributes on average to 7.2% for the stronger one and 9% for the weaker one. The ALL consistency level requires higher nbOps compared to Quorum while both have zero/low readRepairIO and thus according to Formula (6) ALL has a relatively higher storage cost contribution in contrast to Quorum (e.g., it is 7% for Quorum and 7.5% for ALL). Moreover, although the nbOps is smaller for ONE and TWO compared to ALL and Quorum, the increasing number of readRepairIO increases the storage cost. Furthermore, as the cost of readRepairIO is proportional to the rate of stale reads, ONE has higher storage cost contribution in contrast to TWO.

In summary, the read repair function — ensuring that all outdated replicas become up to date —plays a very important role in determining the cost of storage with different consistency levels.

Network cost has also relatively a lower contribution to the total cost for stronger consistency (ALL and Quorum) compared to weaker consistency (ONE and TWO): it contributes on average to 0.175% for the stronger one and 0.275% for the smaller one. The ALL consistency level requires higher interDcRep compared to Quorum (higher number of involved replicas as well as Quorum always tends to answer the requests by involving the most close replicas "within the same datacenter if possible") while both have zero/low IMechTraffic and thus according to Formula 5.11 ALL has a relatively higher network cost contribution in contrast to Quorum. Moreover, although the interDcRep is smaller in for (ONE and TWO) compared to (ALL and Quorum) but the increasing size of IMechTraffic — due to the high rate of stale reads — increases the network cost. Furthermore, as the cost of IMechTraffic is proportional to the rate of stale reads, ONE has higher storage cost contribution in contrast to TWO.

Observation 2 Stronger consistency guarantees cause a higher contribution to instances cost due to the high latency, and a relatively lower contribution to both the storage and network cost as it avoids the extra cost caused by the read repair function.

Results on Amazon EC2

Figures 5.4 and 5.5 support our earlier findings and observations with Grid'5000. The total cost variation in Amazon is lower than in Grid'5000, because of the more powerful machines and the lower cross-sites latency.

As expected, and as shown in Figure 5.4, the total monetary cost decreases when degrading the consistency level. The cost reduces from $32.39 — when the consistency level is set to *ALL* — to $23.23 when the consistency level is *ONE* (i.e., weak consistency reduces the cost by almost 28%). This result was expected as lower consistency level involves fewer replicas in the operations, and thus maintaining low latency, less I/O requests to the storage devices, and less network traffic in general (the runtime of WorkloadA varies from 2 hours to 3 hours and 33 minutes according to the consistency level). This cost reduction, however, comes at the cost of a significant increase in the stale reads rate: as shown in Figure 5.4 79% of the

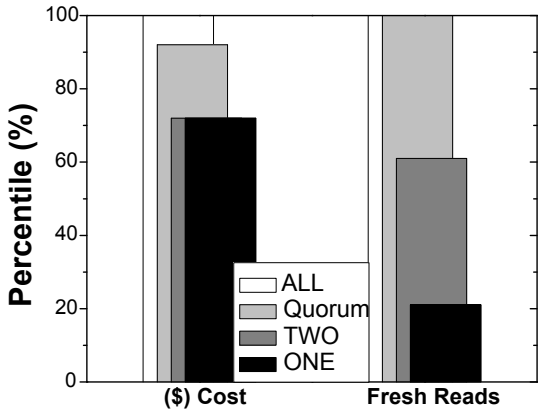


Figure 5.4: Monetary Cost and Fresh reads rate on Amazon EC2

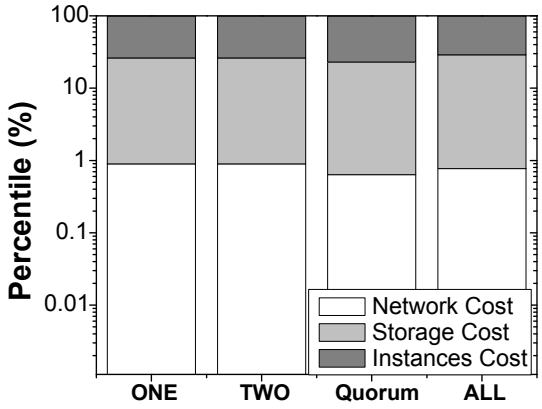


Figure 5.5: Breakdown of the Monetary cost on Amazon EC2 (log scale)

reads are stale reads — only 21% of the reads are fresh reads — when the consistency level is set to *ONE*.

Moreover, the costs of the ONE and TWO levels are the same, although there were significant variations in the running time (2 hours and 1 minutes for ONE and 2 hours and 33minutes for TWO) and also significant variations in the network traffic and storage requests. This is because of the coarse-grained pricing units (per instance hour and per GB storage and per 1 million operations, etc).

Figure 5.5 shows the breakdown of the total cost according to the contributed resources. The instances cost has the higher cost amongst other resources (storage and network): it contributes to almost 74% of the service bill while the storage and network contribute on average to only 25.2% and 0.8%, respectively. This is due to our experiments' scale — number of operations — and the cheap prices of resources (as shown in Table 5.1 the intra communication is free of charges). Moreover, the ratio of the cost of the instances, storage and network to the total cost in Amazon EC2 is different from Grid'5000, because the shorter running time (the high throughput and the powerful machines) which in turn makes the instances cost smaller compared to other resources.

As shown in Figure 5.5, ALL has a relatively higher storage cost contribution in contrast to

Quorum (e.g., it is 28% for ALL and 22% for Quorum). This is because the ALL consistency level requires higher nbOps compared to Quorum while both have zero/low readRepairIO. Moreover, although the nbOps is smaller for (ONE and TWO) compared to (ALL and Quorum) but the increasing number of readRepairIO increases the storage cost. Furthermore, as the cost of readRepairIO is proportional to the rate of stale reads, ONE has higher storage cost contribution in contrast to TWO. The *Network cost* has also relatively a lower contribution to the total cost for stronger consistency (ALL and Quorum) compared to weaker consistency (ONE and TWO): it contributes on average to 0.7% for the stronger one and 0.9% for the smaller one. This cost varies from one consistency level to another according to the number of involved replicas for the stronger consistencies and number of stale reads for the weaker ones.

5.3 *Bismar*: Cost-Efficient Consistency Model

5.3.1 A metric: Consistency-Cost Efficiency

As discussed earlier, data consistency can strongly impact the financial cost of a certain service (i.e., while stronger consistency with high latency implies higher monetary cost of operation as demonstrated in the previous section, the weaker consistency with high throughput causes higher operational cost because of the high rate of stale rate). Consequently, monetary cost should be considered when evaluating the consistency in the cloud [82].

As Cloud Computing is an economy-driven distributed system where monetary cost is explicate and measurable metric [127], we argue that the consistency-cost trade-off can be easily exposed in the cloud. Therefore, we define a new metric — consistency-cost efficiency — that exposes the tight relation between the degree of achieved consistency for a given monetary cost. Our goal is to define a general yet accurate metric to evaluate consistency and thus using this metric as an optimization metric for cloud systems. Accordingly we define the consistency-cost efficiency as the ratio of consistency, measured by the rate of fresh reads, to the relative consistency cost as shown in Formula 5.16.

$$\text{Consistency-Cost Efficiency} = \frac{\text{Consistency}(cl)}{\text{Cost}_{rel}(cl)} \tag{5.16}$$

Where $Consistency(cl) = 1 - stale\ reads\ rate$ and $Cost_{rel}$ is the relative consistency cost with respect to the strong consistency and given by Formula 5.17.

$$\text{Cost}_{rel}(\text{cl}) = \frac{\text{Cost}(\text{cl})}{\text{Cost}(cl\_all)} \tag{5.17}$$

It is important to mention that our metric is designed and can only be applied when strong consistency is not required by an application: we can consider our metric as a system optimization for eventual consistency (i.e., tune the consistency to reduce the monetary cost without violating the application's requirements of fresh read rate).

5.3.2 *Bismar*

We design and implement our approach with the following goals:

Extendable consistency-cost efficiency. Our solution aims at providing consistency guarantees while reducing the monetary cost. Therefore, we propose to use the consistency-cost efficiency metric as an optimization metric: simply by selecting the consistency level with maximum consistency-cost efficiency. Moreover, to meet the diversity of applications requirements (e.g., cost constraint and fresh reads rate constraint), our solution can be easily extended to enable consistency-cost efficiency while favoring either cost or consistency.

Self-adaptive. With the ever–growing diversity in the access patterns of cloud applications along with the unpredictable diurnal/monthly changes in services loads, it is important to provide a self-adaptive approach that transparently scales the consistency level up/down at runtime without any human interaction. Therefore, our approach embraces an estimation model for consistency-cost efficiency that could be achieved with different consistency levels: at runtime, the application's access pattern and network latency are fed to the consistency probabilistic estimation model (we have extended the model in Chapter 4 as will be explained later in this section) in order to estimate the rate of stale data that could be read in the storage system. Furthermore, we use the same information (e.g., access pattern and network latency) along with the predicted stale read rate (i.e., to estimate the number of stale reads) to compute the monetary cost.

Independent of pricing schemes. Our solution targets public cloud and is not limited to any cloud provider in terms of provided services or pricing schemes. The fine-grained monetary cost analysis that is used for cost estimation can be easily adopted to different services and pricing.

Independent of cloud storage systems. Since our solution is implemented as a separate layer at the top of the cloud storage system, it does not impose any modifications to the cloud system code. Our approach, therefore, can be applied to different cloud storage systems that are featured with flexible consistency rules.

Consistency Probabilistic Estimation.

In the previous chapter, we introduced an estimation of the stale reads rate in the system by means of probabilistic computations. This estimation model requires basic knowledge of the application access pattern and of the storage system network latency. Network latency in this case is of high importance, since it is the determinant of the updates propagation time to other replicas. The access pattern, which includes read rates and write rates is a key factor to determine consistency requirements in the storage system.

The probability of a stale read $Pr(staleRd)$, assuming that writes are performed with a consistency level that involves only one replica, is given by Formula 5.18 where rf is the replication factor, and T_p is the average time to propagate an update to other replicas.

$$Pr(staleRd) = \frac{(rf-1)(1-e^{-\lambda_r T_p})(1+\lambda_r\lambda_w)}{rf\lambda_r\lambda_w} \tag{5.18}$$

Given that when the storage system supports multiple consistency levels, the consistency level for read and write operations (cl_r and cl_w respectively) may vary with time. Accord-

ingly, we extend the probability model in Formula 5.18 to consider all the consistency levels for write and read operations that are smaller or equal to the Quorum level, where a Quorum is computed as: $\lfloor \frac{replicationfactor}{2} + 1 \rfloor$. This probability is given in Formula 5.19.

$$Pr(staleRd) = \frac{(rf - (cl_w + cl_r - 1))(1 - e^{-\lambda_r T_p})(1 + \lambda_r \lambda_w)}{rf\lambda_r\lambda_w} \tag{5.19}$$

Efficiency-aware algorithm.

Many applications do not strictly require strong consistency: a consistency optimization solution, therefore, can be introduced to improve system throughput, latency and monetary cost. To achieve this goal we consider our metric as an optimization metric as shown in the following algorithm.

Algorithm 2: Cost–Efficient Consistency

```
while true do
    for cl ∈ CLs do
        Compute Cost_rel(cl) Compute Consistency(cl)
        Compute Consistency(cl)/Cost_rel(cl)
    end
    Choose cl∈ CLs for Max[Consistency(cl)/Cost(cl)]
end
```

At runtime, our system feeds the efficiency-aware algorithm with data related to the system read/write rates along with the network latency. These data are used by the consistency probabilistic estimation model to compute the expected achieved fresh reads when using different consistency levels. The relative monetary cost is also computed according to the system configuration and the stale read estimation. So the algorithm Algorithm 2 selects the consistency level that offers the most equitable consistency–cost tradeoff (the maximum consistency-cost efficiency value).

5.4 Experimental Evaluation

We have built our approach as a separate layer on top of *Apache Cassandra-1.0.2* [85]. The core of this layer consists of two modules. Both modules were implemented in Python 2.7. *The monitoring module* collects relevant metrics (data) needed for our approach of the storage system's information. The data is further communicated to the *dynamic consistency module*. An estimation of consistency-cost efficiency is computed — according to the estimated stale reads rate and the monetary cost (instance, storage and network cost)— and then compared in order to provide a cost efficient consistency level for the running application at that point of time. Later in this section, we present our detailed evaluation of our consistency-cost efficiency metric and the *Bismar* prototype using Cassandra on Grid'5000 testbed. We also use YCSB to run WorkloadA (we have used the same testbed and WorkloadA described in Section 5.2.3). In order to present the dynamicity of the system (i.e., the variation of throughput

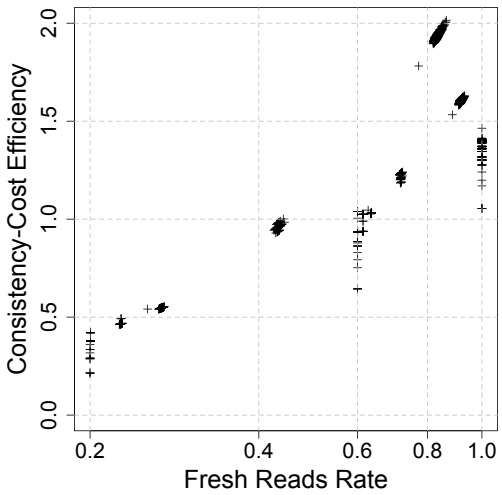


Figure 5.6: Consistency–Cost–Efficiency Effectiveness

and the read/write rates during the runtime), we ran the workload, varying the number of threads starting with 1 thread, then, 50, 20, 7 and finally, 30 threads.

5.4.1 Consistency–Cost Efficiency

In order to validate our metric, we collect samples when running the same workload (with different consistency levels), varying the number of client threads, and thus exhibiting different access patterns. Figure 5.6 shows the results where each point represents a different access pattern. Higher consistency-cost efficiency values are associated with high rates of fresh reads (around 80%). This indicates the effectiveness of our metric: it is designed to achieve the best price without violating the consistency (we consider the 80% fresh reads as acceptable consistency).

5.4.2 Monetary Cost

Figure 5.7 shows the monetary costs of running the workload with the three static consistencies (ONE, TWO and Quorum) and with our dynamic adaptive approach. As expected, ONE exhibits the lowest monetary cost but at the cost of fresh reads. Our experiments also show some interesting results: *Bismar* achieves lower cost in contrast to the consistency level TWO. Since *Bismar* always selects the consistency level with the highest consistency-cost efficiency to adopt to the workload dynamicity, *Bismar* adopts the consistency level ONE for almost 70% of its running time while it adopts the consistency level Quorum for 30% of its running time as shown in Figure 5.8. As a result, the cost reduction when running with ONE overcomes the cost increase when running with Quorum. Since *Bismar* targets applications that do not require strong consistency, we consider *Bismar* as an approach for optimizing eventual consistency on cloud platforms. It improves the monetary cost of services while maintaining acceptable rate of fresh reads. It is also interesting to compare the cost reduction and performance improvement by *Bismar* in contrast to the Quorum consistency level.

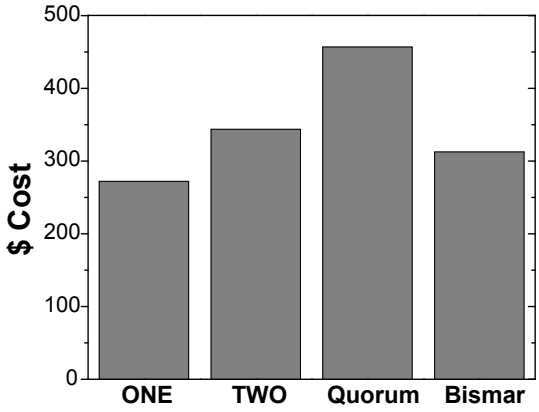


Figure 5.7: *Bismar :* Monetary Cost

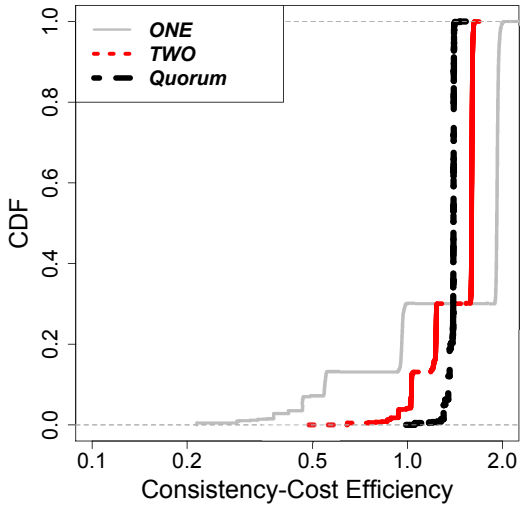


Figure 5.8: Consistency–Cost–Efficiency Distribution

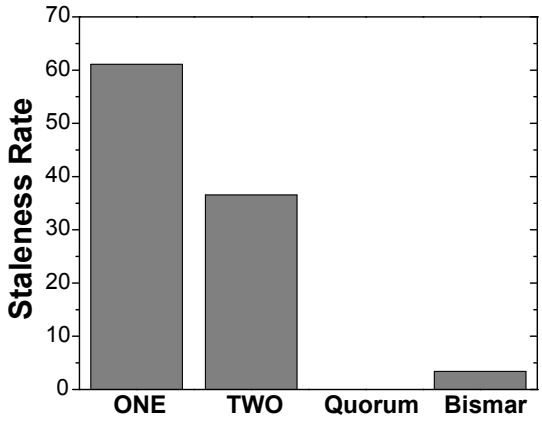


Figure 5.9: *Bismar :* Staleness

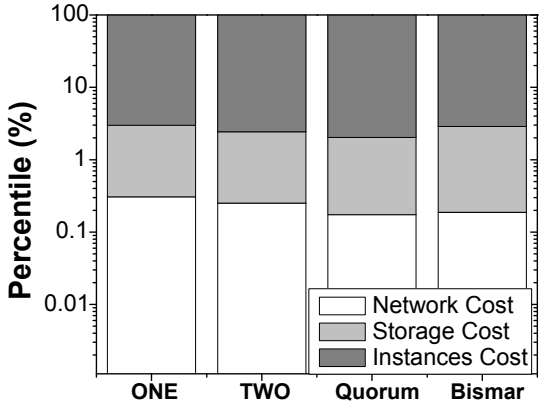


Figure 5.10: *Bismar :* Cost breakdown

As shown in Figure 5.7, *Bismar* reduces the monetary cost by almost 31.5% in contrast to Quorum level (From \$456 to \$312). The cost reduction is mainly due to the performance improvements (*Bismar* improves the overall response time by almost 32.2%).

5.4.3 Staleness vs. monetary cost

Figure 5.9 shows the stale reads rates caused by different consistency approaches. It is clear that static levels *ONE* and *TWO* produce higher stale reads rate: 61% of the reads where on stale data with ONE and 36% with TWO. Moreover, the Quorum consistency level returns always up-to-date data (i.e., stale reads rate is 0%) because at least one replica with the freshest data should be in the Quorum. *Bismar* however, returns very small portion of stale reads (only 3%), but with very important money saving (31.55% cost reduction compared to Quorum). The 3% stale reads is considerably reasonable for many applications.

5.4.4 Zoom on resource cost in *Bismar*.

Figure 5.10 shows the breakdown of the total cost according to the contributed resources for different consistency levels and *Bismar*. As shown and discussed earlier in Section 5.2.3, the

instance portion of the total cost increases with upgrading consistency while the portion of both the storage and network costs increase with degrading the consistency level. However, the aforementioned observation is also applied on *Bismar*: comparing *Bismar* against Quorum, we notice that instance cost portion in *Bismar* is lower than in Quorum. Furthermore, we observe that the portion of both the storage and network costs in *Bismar* is higher than in Quorum. This can explain why the cost reduction was only 31.5% while the performance improvement was 32.2%: because of the adversary impacts of the storage and network costs in *Bismar*. Moreover, we observe that the portion of both the storage and network costs in *Bismar* is higher than in all static consistency schemes, because *Bismar* combines both the high number of requests when adopting a higher consistency level and also read repair cost when stale reads is detected when *Bismar* adopts lower consistency level.

5.5 Discussion

With the explosive growth of data size and availability requirements of services in the cloud along with the tremendous increase in users accessing theses services, geographically distributed replication has become a necessity in the cloud storage [63][40][48]. At such scale, the strong consistency suffers of high latency and thus violating both the performance and availability requirements. Cloud storage is therefore evolving towards eventual consistency. Eventual consistency has been extensively exploited in literature and commercial products such as Dynamo [49] in Amazon S3 [13] and Amazon DynamoDB [7], Cassandra [85] in Facebook [59] and Yahoo! PNUTS [44] in Yahoo!. While the most of the work in literature have been dedicated to either measuring the actual provided consistency in cloud storage platforms [126, 17, 29], or on adaptive consistency tuning in cloud storage systems [82, 111, 128, 87] in order to meet the consistency requirements of applications and reduce the consistency violation. Despite our work being focused on the monetary cost, a key difference between our work and their work is that we are seeking an adaptive consistency approach, which is at the same time cost efficient and does not violate the applications needs.

A closely related work on improving the monetary cost of consistency in the cloud is [82]. *Kraska et al.* propose consistency rationing: an automatic approach that adapts the level of consistency at runtime considering the performance and monetary cost. The authors define consistency levels at data level (i.e., categorizes the data into three types and provides a different consistency treatment for each category). Consistency rationing at data level may incur additional meta data management overhead when the data size is large, our work therefore is at a transaction level: our adaptive tuning approach chooses the number of replications involved in an operation considering the best trade-off between the consistency level and monetary cost. The results discussed in our work complement Kraska's work: monetary cost-oriented consistency approach at transaction levels to complement their work at data level.

With respect to monetary cost in cloud systems, a number of studies [104, 50] have been dedicated to measure the cost of adopting the pay-as-you-go cloud in terms of monetary cost, performance, and availability. Some studies [127, 62, 76] have reported on the cost variations and fairness in the cloud. Many recent studies concentrate on monetary cost improvements of cloud services through reducing the virtualization interference [77], using spot instance or levering the public cloud using free resources such as desktop grid [42][89].

In contrast, this work investigates the interplay between economic issues and the consistency design and implementation.

5.6 Summary

In this study, we investigate the monetary cost of consistency in the cloud. Our detailed analysis and study has revealed a noticeable monetary cost variation when different consistency levels are used. Strong consistency levels have a tendency to consume more resources at increasing monetary costs while weaker levels reduce the monetary cost at an increasing rate of stale reads. In order to fully understand the impact of the different consistency levels on the monetary cost and the rate of fresh reads in the cloud, our *consistency-cost efficiency* metric have demonstrated its proficiency. We have shown how this new metric reflects the tight relation between the level of consistency used and the monetary cost in the cloud in the form of a ratio that can be used in decision making. Moreover, and in order to deal with the dynamicity of cloud workloads, our adaptive approach *Bismar* leverages this metric to provide cost–efficient consistency. Our experimental evaluations on a large cluster of Cassandra cloud storage have demonstrated that such an adaptive approach leads to important cost cuts (up to 31%) at an acceptable consistency for this type of workloads.

Chapter 6

Consistency vs. Energy Consumption: Analysis and Investigation of Consistency Management impact on Energy Consumption

Contents

AFTER addressing the impact of consistency management on performance (in Chaptrer 3) and monetary cost (in Chapter 4), in this chapter we investigate its impact on energy consumption in the cloud. Within today' Big Data scales, the power management and the energy consumption in the datacenter are highly important issues. In this context, we introduce a first study that focuses on exploring the energy consumption and the power usage when different consistency models are used. Accordingly, we investigate how to leverage such a study as to reduce the overall energy consumption of the storage cluster.

6.1 Motivation

Energy consumption within data centers is increasing at alarming rates [55, 58, 47]. It is reported that the power usage within the Facebook datacenter in Prineville has more than doubled in one year [58] while Hamilton [47] has estimated that, in 2008, the power usage of the servers and the cooling units has exceeded 40% of the total cost for data centers. As a result, power management in data centers has become an extremely important issue. A better management of power and energy is necessary in order to protect our environment, as well as to reduce the monetary cost. In the era of Big Data, a huge part of the datacenter activity is related to data-intensive applications. Therefore, optimizing energy consumption within distributed storage systems is becoming a priority.

Nowadays storage systems provide novel designs and data structures in order to deal with the data tsunami. In this context, as shown in Chapter 3, many modern storage systems trade consistency for better performance, availability and sometimes monetary cost as presented in Chapter 5. This has resulted in various consistency models being introduced and implemented. The impact of these consistency models, including strong and eventual consistency, on performance and availability has been widely studied. However, no studies have focused on their impact on the very important matter of energy consumption, in particular with today large scales.

In this study, our primary goal is to investigate and highlight the unexplored impact of consistency management on energy consumption. Accordingly, we conduct series of extensive evaluations in order to analyze the power usage patterns, and to investigate areas that may be further explored as to save energy. In this context, the aim of our work can be summarized in the two following goals:

Analysis of the impact of consistency levels on energy consumption. Investigate and study how the two main popular consistency models: eventual consistency (with its different levels) and strong consistency influence the energy consumption of a distributed storage system. The study focuses on the investigation of the power usage, the provided performance, and the resource usage exhibited with each model.

Investigation of adaptive configurations to save energy with regard to consistency. The second goal is to leverage the analysis study of consistency impact on energy consumption as to introduce best practices and storage cluster configurations that result in energy savings.

6.2 Insight into Consistency–Energy Consumption Tradeoff

In this first step, we investigate, experimentally, the tradeoff between consistency and energy consumption. Accordingly, further analysis is provided as to build in-depth picture of consistency models impact on the energy consumption of the storage cluster.

6.2.1 Tradeoff Practical View

Experimental Tools. First, we present the different tools and the micro benchmark used in our experimental evaluation.

YCSB. As previously presented in Chapters 4 and 5, The Yahoo! Cloud Serving Benchmark (YCSB) [137] is an ideal micro–benchmark that exhibits real cloud features and workloads. Moreover, YCSB is already adapted to be used with multiple open–source cloud storage solutions, including Apache Cassandra. In this experimental evaluation, we use YCSB-0.1.4 with a slightly–modified Java client (as to tune the consistency level) that interacts directly with the Cassandra Cluster.

Dstat: Versatile resource statistics tool. Dstat [52] is a versatile tool for monitoring resource usage of the system. Dstat replaced multiple other tools, and allows therefore the monitoring of various resources including CPU, Memory, Disk, Network among others. In our experiments, we use Dstat, mainly, to monitor the CPU usage and the memory usage within the storage cluster nodes.

PDU: Power Distribution Unit. A PDU is a hardware device that distributes electrical power within a rack or a cluster to the different computing nodes, networking devices and potentially other hardware equipment. Moreover, PDUs keep track of the distributed power to the different nodes and allow the monitoring of power usage. In this context, we use PDUs as power monitoring devices where the monitoring controllers use the Simple Network Management Protocol (SNMP) [39].

Experimental Setup. Our experiment sets have been conducted on the French Grid and Cloud testbed Grid'5000 [79]. Grid'5000 federates 10 sites in France and Luxembourg with a total number of cores that exceeds 8000. We deployed Cassandra on 39 nodes on the *Graphene* cluster in the *Nancy* site. The Graphene cluster consists of 144 nodes. However, only 40 nodes are attached to PDUs. Every node is equipped with a 320 GB disk space, 16 GB of memory size, and 4-cores Intel Xeon CPUs. The nodes are connected via Gigabyte Ethernet. Two types of PDUs are installed. The first type is an EATON PDU (from the EATON corporation [54]) that insures the power distribution to 40 nodes in the Graphene cluster. The remaining PDUs are 2 APC PDUs (provided by APC [24]). Every APC PDU consists of 20 outlets each connected to a computing node, thus allowing power measurements node by node.

In our experiments, we have used Apache Cassandra-1.1.4 with a replication factor of 5. As introduced in Chapter 3, Cassandra is a very popular system with a versatile consistency usage and efficient memory usage that fits our case study. We have initially run YCSB to load 2 Million keys with roughly 10 GB of data after replication. We have run a reads–updates–heavy workload that consists of 20 Million operations with a read/update ratio equals to 60/40. Moreover, we have run the workload with three values of client threads number: 20, 70, and 100.

Energy Consumption Evaluation

Figure 6.1 shows the energy consumption when variating the consistency level: One (basic eventual consistency), Quorum (quorum–based eventual consistency), and All (strong consistency). The main observation is that energy consumption increases when increasing the consistency level demonstrating the existence of an inevitable *consistency–energy saving tradeoff*. For the different numbers of threads (20, 70, and 100), the consistency level One consumes significantly less energy than the Quorum-based level and the strong consistency

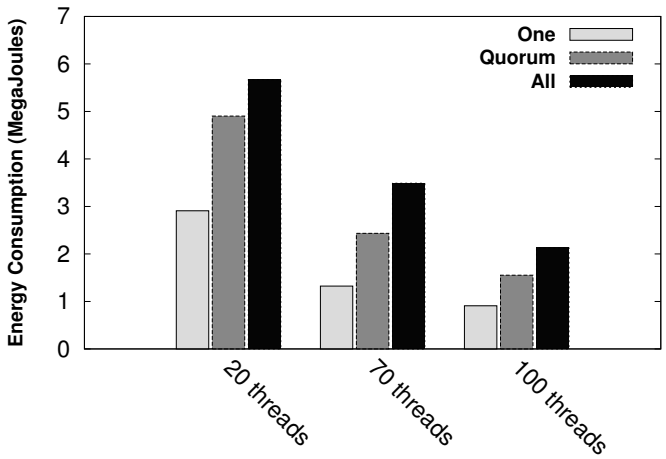
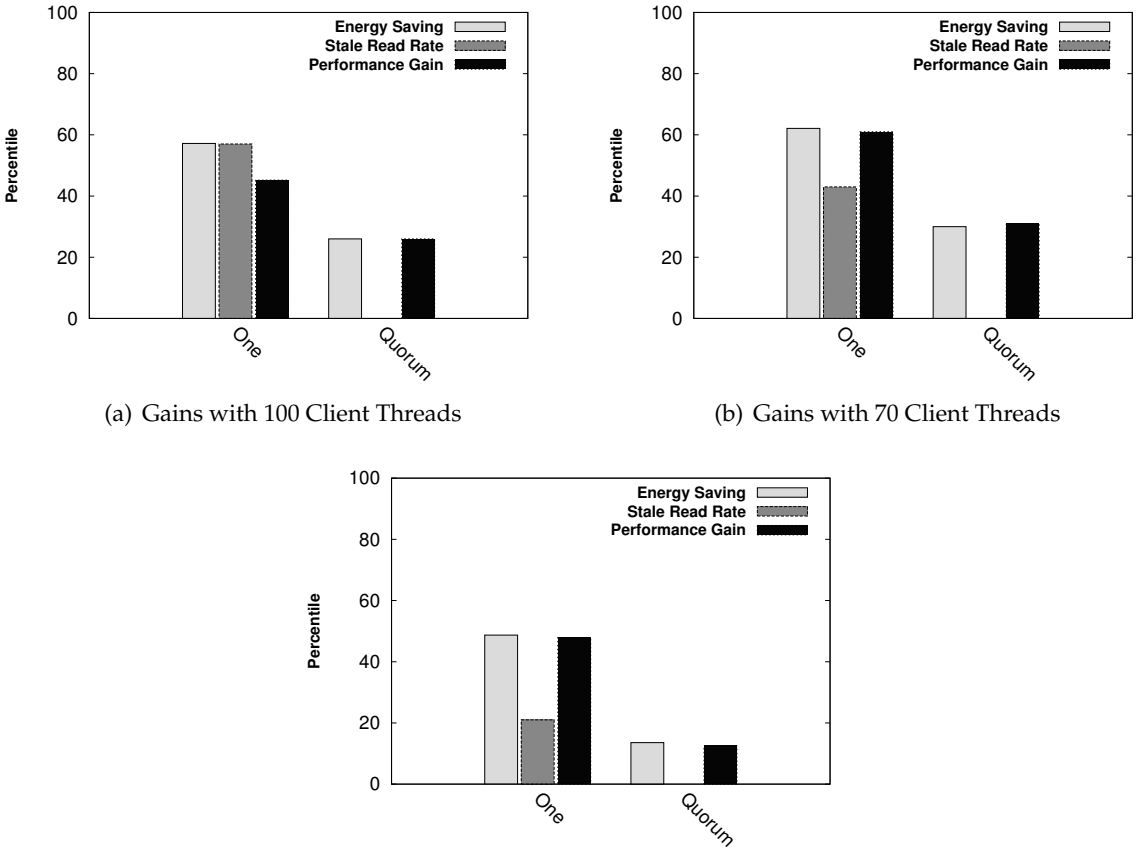


Figure 6.1: Energy Consumption with various consistency levels

(the All level). This confirms our expectations, since the weaker consistency levels use less resource for a given operation –involving smaller number of replicas– than the stronger consistency levels, and have a performance edge that reduces the execution time. For instance, when the number of client threads is 70, the One level saves up to 62% of energy compared to the strong level (with its synchronous replication). An additional observation is that the energy consumption decreases when the number of client threads increases for all consistency levels. This is explained by the fact that increasing the number of client threads enhances the throughput (the performance) of the storage system (Cassandra). Therefore, executing the same workload with an enhanced performance results in a shorter runtime and consequently reduces the energy consumption.

Energy Savings, Data Staleness, and Performance Gains

Besides the energy consumption of different consistency levels, and for fair evaluation, we highlight both the level of consistency guaranteed (using the estimation of stale reads rate) and the performance provided. Energy consumption is a very important metric in nowadays data centers. In this context, it should be considered when managing consistency. In Figures 6.2(a), 6.2(b), and 6.2(c) we show the energy saving, the stale read rate, and the performance gain of the two levels One and Quorum when the number of client threads is 100, 70, and 20 respectively. With 100 and 70 client threads, the energy saving of the One level exceeds 57% and 62%, respectively in comparison with strong consistency. These significantly high saving rates are due to the usage of few resources (Only one replica is involved in the operation) under heavy throughputs exhibited by the high number of client threads. In contrast, the strong consistency level (All) involves all replicas in operations under heavy accesses, which results in high energy consumptions. Similarly, the performance is respectively 45% and 61% better since only one replica is involved with the One level generating less traffic in the network and exposing higher throughput to the client. However, these energy savings come with an inconsistency overhead. The stale read rate of the One level is significantly high (57% and 43% respectively). The main reason for these high rates is the ex-



(a) Gains with 100 Client Threads

(b) Gains with 70 Client Threads

(c) Gains with 20 Client Threads

Figure 6.2: Energy saving, stale read rate, and performance gain of weak consistency levels

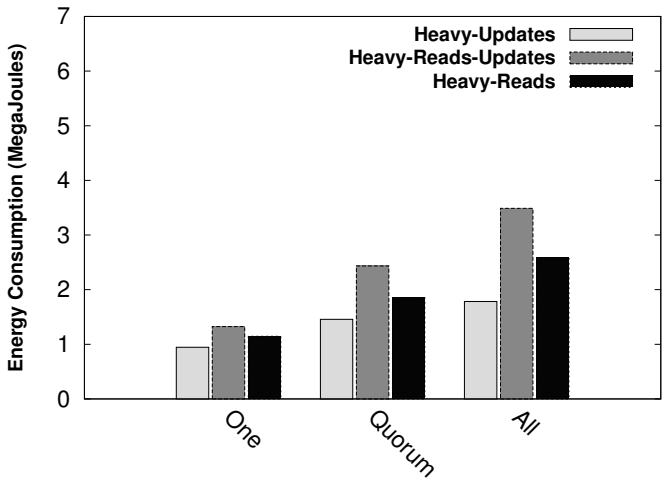


Figure 6.3: Read and Write rates impact on Energy consumption

hibited high read rate and the high write rate generated by multiple threads accessing data at the same time. Alternatively, the Quorum-based level insures that all data read is fresh since the replica with the latest update is always included in the quorum. Meanwhile, the quorum level exhibits 27% and 30% of energy savings when the number of client threads is 100 and 70, respectively.

When the number of client threads is relatively small (20 threads), the throughput in terms of served operations per second is smaller. Figure 6.2(c) shows that in this case, the energy saving is still significant (roughly 49%) while the rate of stale reads is relatively acceptable (21%) with the basic One level. The Quorum level provides higher consistency guarantees (with 0% stale read rate) while reducing the energy consumption by 14% and enhancing performance by 12%. The results show therefore that strong consistency comes with high energy consumption (In addition to the performance overhead and the monetary cost shown in Chapters 4 and 5). The general belief was that when performance and availability are not a priority, it is better to implement a strong consistency. However, using this consistency model when it is not strictly needed might introduce a heavy impact related to the energy consumption (in addition to the monetary cost) within the datacenter. Therefore, applications for which update conflicts are efficiently handled (at the application or the system level) can benefit from the Quorum level when the read rate and write are high; and the One level when these rates are low in order to save energy without consistency violations.

6.2.2 Read/Write Ratio Impact

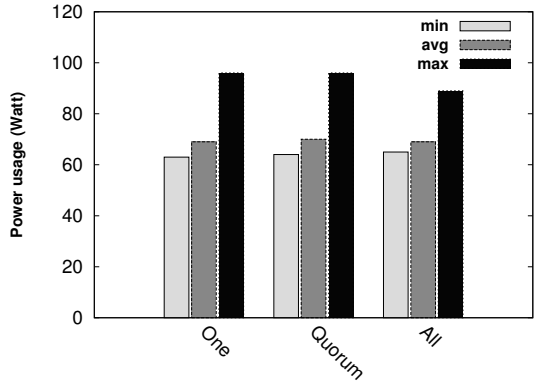
In this section, we focus on investigating the impact of read/write ratio. Figure 6.3 shows the energy consumption —where the number of client threads equals to 70— of three different workloads: the first one is an updates–heavy workload with a read/write ratio of 20/80, the second is a reads–updates–heavy workload with a read/write ratio of 60/40, and the last workload is a reads–heavy workload with a read/write ratio of (80/20). Surprisingly, we can observe —independently of the selected consistency level— the updates–heavy workload is the workload with the least energy consumption. Moreover, the reads–updates–heavy work-

load is the one with the highest consumption exceeding that of the reads–heavy workload. In order to understand such unexpected results, we need to analyze the internal mechanisms of the Cassandra storage system. Cassandra, much like many NoSQL data stores, is optimized for write throughput. This is mainly because write operations are considered as extremely important and should always be available at a low latency (For instance Amazon *Shop Cart* should always be available for new purchases, otherwise, unavailability will result in a loss of money). Therefore, write latency within Cassandra is very small since a write success is issued when data is written to the log file and the memory (not the disk). In this context, the small consumption of the updates–heavy workload is due to writes small latency. Write latency is even smaller than the read latency. Data in Cassandra is written to *memtables* in memory and flushed later to *sstables* that are written sequentially to disks. The sstables might however contain data rows that diverge overtime. In order to handle this issue, Cassandra implements a *Compaction* process in the background to merge sstables. This in turn, introduces extra latency when fetching data for read operations and thus explains why reads–updates–heavy workload consumes more energy. In the worst–case scenario, a reads–updates–heavy workload results in a more frequent *compaction*, because of the high number of update operations, and therefore it further increases the read latency in comparison to the reads–heavy workload. When the updates number is high more data is written to the sstables that grow high very fast at high probability of diverging rows (always because of potential updates to the same rows) thus increasing the frequency of the compaction operation that affect the read operations (which are far more numerous compared to the updates–heavy workload).

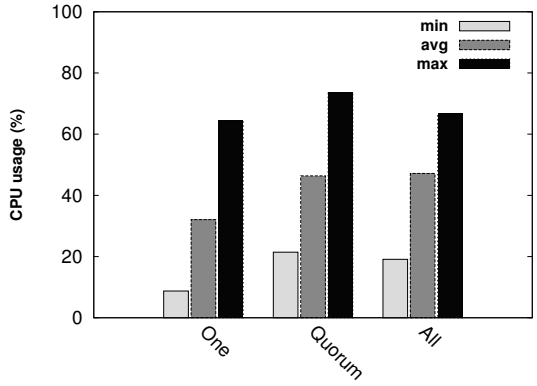
6.2.3 Nodes Bias in the Storage Cluster

In figures 6.4(a), 6.4(b), 6.4(c), we show the power usage, the CPU usage, and the memory usage, respectively for the One, the Quorum, and the strong consistency levels with a reads–updates–heavy workload and 70 client threads. The experimental results indicate that the average power usage (the average of power usage of all nodes at all time periods) differs slightly between the consistency levels. However, the gap between the max value and the min value is relatively large, and largest with the One level, which is an indicator of a potential variation of power usage between nodes. The average of CPU usage on the other hand, is higher with the stronger consistency levels. This is mainly because CPU is more active since more replicas (and thus nodes) are involved during access operations. Moreover, there exists a huge gap between the max usage and the min usage, in particular for the One level where the min usage is roughly 8% (indicating that the node is almost idle) and the max value is approximately 64%. This gap is higher with CPU usage than the power usage because the latter has a steady consumption portion of roughly 44 Watt (even at the idle time). In contrast to power and CPU usage, the memory usage is steady for the different consistency levels with a minimal gap between the minimum usage and the maximum usage. These results are explained by the fact that Cassandra keeps data in memory in *memtables*. Therefore, no matter the consistency level used or the number of times the node is solicited, the size of data kept in memory is the same.

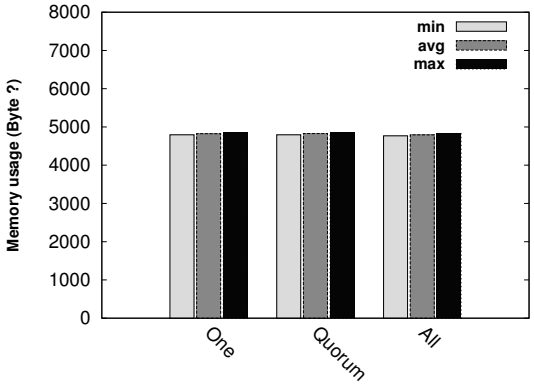
In order to further highlight the variation of power usage and CPU usage and the bias of the cluster nodes, we compute the *coefficient of variation* for the average values per node. The *coefficient of variation* CV is computed as $CV = \frac{Standard\ Deviation}{Mean}$. Figure 6.5 shows the CV for



(a) Power Usage



(b) CPU Usage



(c) Memory Usage

Figure 6.4: Minimum, Average, and Maximum usage of Power, CPU, and Memory of the storage nodes

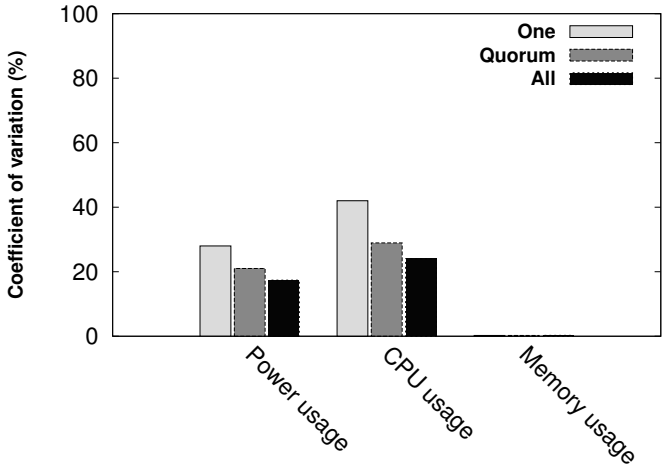


Figure 6.5: Coefficient of Variation in resource usage of the storage nodes

power usage, CPU usage, and memory usage. The results clearly demonstrate the existence of variation of power usage between the storage cluster nodes that is highest with lowest consistency level. Access operations that use low consistency levels involve only a subset of replicas, which results in a higher power usage for nodes hosting those replicas compared to other nodes. With the strong consistency level, all replicas are involved in the data access operations exhibiting a smaller coefficient of variation. The small observed variation is mainly due to the non-uniformity of key distribution (even with an equal data range partitioning) which is a known problem for systems based on distributed hash tables such as Cassandra. However, the overall consumption variation of power usage remains smaller than the variation observed with the CPU usage (Variation exceeds 42% for the One level against roughly 28% for power variation). The difference is explained by the fact that in Cassandra storage, much like many eventually–consistent storage systems, the nodes within the cluster are peers and have equal responsibilities and host equal ranges of data (assuming a uniform distribution of data keys). Therefore, nodes might consume extra power for cluster management and request routing even without being solicited for serving reads and writes from their replicas. The memory usage on the other hand, exhibits no variation for the reasons previously mentioned (nodes keep the same size of data in memory). The aforementioned observations show clearly that energy is being wasted on nodes with little activity (For instance one node has an average cpu usage of only 8% with the One consistency level). One solution is to assign fewer tasks to the nodes —without powering them down to insure data durability and persistence — that do not get frequent requests for their hosted replicas. In the next section, we investigate the possibility of avoiding such situations.

6.3 Adaptive Configuration of the Storage Cluster

6.3.1 Reconfiguration Approach

The results presented in the previous section show a variation in CPU usage and power usage between the different nodes of the storage cluster, in particular with low consistency levels. As previously explained, the problem is caused by treating all the nodes as equals while not all of them are involved in access operations resulting in energy waste on inactive nodes. In order to overcome this situation, we propose a new cluster configuration of the storage cluster. We divide the storage nodes into two pools: the *warm pool* and the *cold pool*.

The warm pool. This pool includes the cluster nodes that are most active and highly consuming. Nodes within the warm pool are assigned with more responsibilities and larger data ranges to host as shown in Figure 6.6. Data partitioning in this case is reconfigured in order for warm nodes to be responsible of larger number of keys. In Cassandra, this is accomplished by assigning more tokens in the ring to these nodes. Moreover, the warm nodes are exclusively responsible of handling client requests. Client requests are not directed to nodes within the cold pool. Upon a request arrival, a warm node will determine which node hosts data for the requested key. Since larger data ranges are assigned to warm nodes, the probability that the data–hosting node will fall within the warm pool is high.

The cold pool. In contrast to the warm pool, the cold pool includes nodes that are not highly active. Therefore, in this configuration, we put those nodes on a low consumption

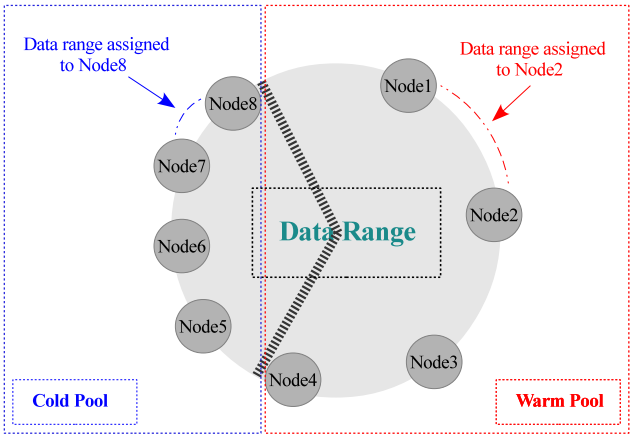


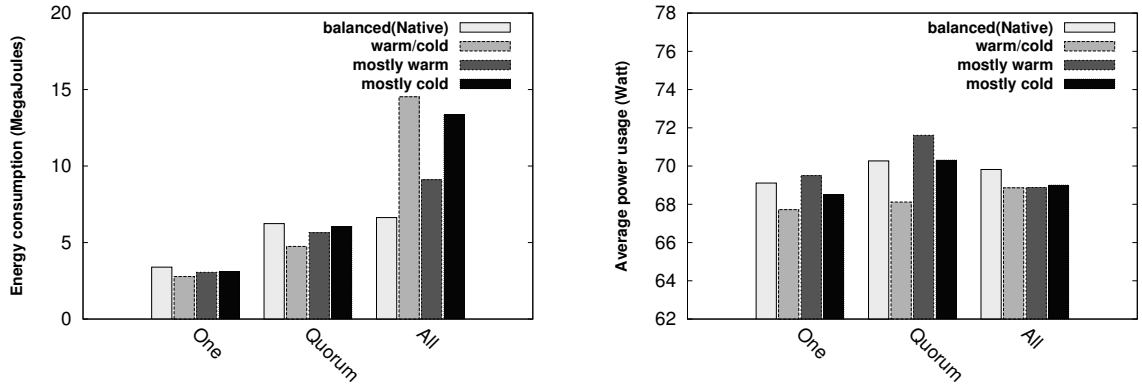
Figure 6.6: Data Distribution for warm/cold Pools: warm nodes are assigned with more tasks and larger data ranges than cold nodes

mode (for instance using DVFS *Dynamic Voltage and Frequency Scaling* [117] technique to lower the CPU frequency of these nodes due to the low usage) and charge them with minimal tasks. Therefore, data ranges assigned to the cold pool are smaller in order to reduce their solicitation as shown in Figure 6.6. Nodes in this case, are involved in operations when data fall in their (small) ranges, to respond to other replicas request with strong consistency levels, and with internal mechanisms of the storage system (Read Repair mechanism, and the Gossip protocol for Cassandra). As a result, with eventual consistency, the probability that the request will be served exclusively within the warm pool is high, especially when data keys are created carefully (in an uniform manner).

This configuration of Warm and cold pools should be dynamic and adaptively reconfigured. Cold nodes could join the warm pool during peak load times and warm nodes could join the cold pool during non–busy periods. Moreover, the reconfiguration must consider the dominating consistency level used in the workload to adapt properly. The low levels introduce more variation between the nodes than the strong ones and provide better performance. As a future plan, we intend to build an adaptive reconfiguration approach that dynamically readapts the warm and the cold pool sizes to efficiently serve data while reducing the energy consumption.

6.3.2 Experimental Evaluation

Setup and Storage Configurations. In order to investigate the cluster reconfiguration impact on energy consumption, we have run a reads–updates–heavy workload (with a read-/write ratio of 60/40) and 70 client threads. Moreover, we have used four storage cluster configurations where every node in the warm pool is assigned twice the size of data range assigned to a node in the cold pool:



(a) Energy Consumption

(b) Average Power Usage

Figure 6.7: Energy consumption and average power usage of different configurations

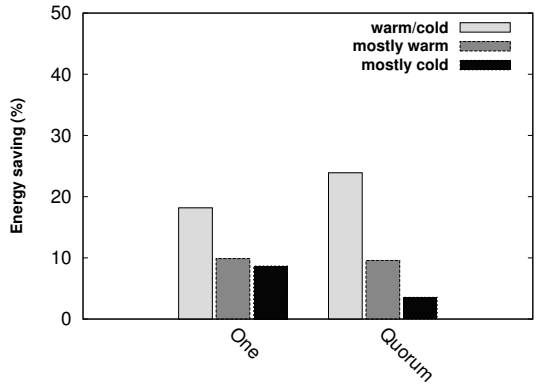
Balanced. This is the native configuration with one pool of nodes that share the same tasks and host the same data size.

Warm/cold. This configuration divides the nodes set into two equal subsets one assigned to the warm pool and the other to the cold pool.

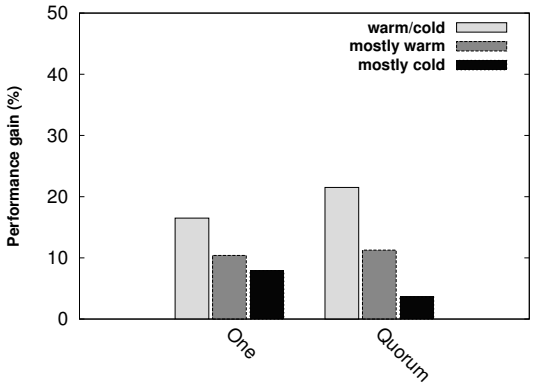
Mostly warm. 2/3 of nodes are assigned to the warm pool within this configuration and only the remaining 1/3 of nodes is assigned to the cold pool.

Mostly cold. 2/3 of nodes belong to the cold pool while only 1/3 of nodes belong to the warm pool.

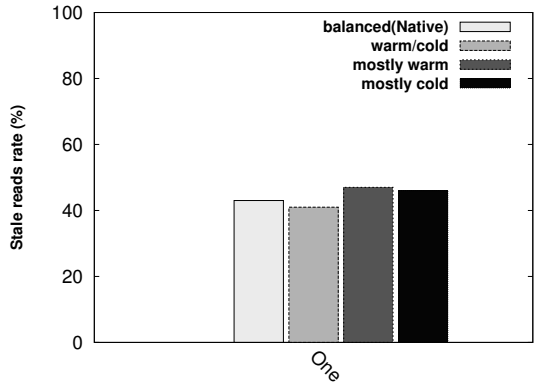
Energy Consumption and Power Usage Evaluation. Figures 6.7(a) and 6.7(b) show the overall energy consumption, and the average power usage, respectively, of our applied four configurations. Both the energy consumption and average power usage are lowest when the warm and the cold pools are of equal size (warm/cold configuration) with eventual consistency (The One and the Quorum consistency levels). Moreover, both the mostly warm and the mostly cold configurations consume less energy than the balanced configuration. This clearly shows that for eventual consistency a balanced configuration with a balanced data distribution client requests is not the best solution to reduce the energy consumption. Since only a subset of replicas is involved in access operations, some nodes are busier than others. Therefore, energy is "wasted" on "lazy" nodes. The average power usage for the mostly warm configuration is higher than the balanced one because of the high number of nodes in the warm pool. However, the overall energy consumption is smaller because of the performance gains (as shown in Figure 6.8(b)) that result in a smaller execution time. Moreover, the average power consumption for both the mostly warm configuration and the mostly cold configuration is higher than the average of warm/cold configuration, which explains, in addition to the performance gains, why energy consumption is lower with this configuration. In contrast to eventual consistency, strong consistency achieves, by far, lower energy consumption when the cluster is balanced and where nodes share roughly the same amount of data and tasks. Strong consistency dictates that all replicas must be involved in



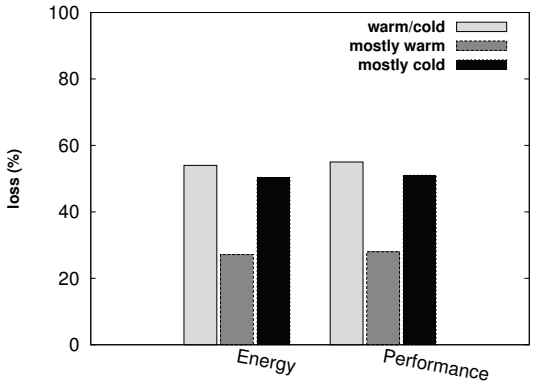
(a) Energy savings of unbalanced configurations



(b) Performance gains of unbalanced configurations



(c) Stale read rate of the One consistency level



(d) Energy and Performance loss with strong consistency

Figure 6.8: Gains, stale read rate, and loss of consistency levels with different configurations

access operations. Therefore, a balanced configuration is considered as a one warm pool where all the nodes are solicited either to serve the client requests directly or to perform an action on its replica (send the data or the metadata for read operations or commit the update for the write operations). On the other hand, the average power usage is higher with the balanced configuration compared to the others. This is because of two reasons. First, for the non-balanced configurations, nodes in the cold pool have a low power usage caused by the drop of overall average of resource usage. Second, the performance loss (as it will be shown in Figure 6.8(d)), in the form of throughput decrease, lowers the resource usage of the nodes, which results in a smaller power usage. However, the performance drop results in a higher execution time and thus, higher overall energy consumption.

Gains Analysis. In Figures 6.8(a), and 6.8(b), we show the energy savings and the performance gains respectively, compared to the balanced configuration (for eventual consistency). The configuration with equally–sized warm/cold pools achieves the highest savings that reach up to 19% of the already low consumption when the level of consistency is One, and up to 23% for the Quorum level. For the type of the workload applied, the configu-

ration where most of the nodes belong to the cold pool is the one with the lowest saving (8% for the level One and only 3% for the level Quorum). Similarly, the performance gains are highest with the equally divided pools configuration and lowest with the mostly cold configuration. Moreover, Figure 6.8(c) shows the stale reads rate of all the configurations with the consistency level One. Both energy savings of the mostly warm and the mostly cold configurations come at the cost of adding a very small portion of stale reads to the one of balanced configuration (4% for the mostly warm and 3% of the mostly cold). Interestingly, the warm/cold configuration exhibits the lowest stale reads rate (41% vs. 43% for the balanced configuration). Therefore, and for this type of workloads, the warm/cold configuration is the *best* choice for *eventual* consistency providing the highest energy saving, the highest performance, and the lowest stale reads rate.

The best–fit configuration when adopting strong consistency is the balanced configuration as shown earlier. In this context, Figure 6.8(d) shows both the energy waste and the performance drop of the unbalanced configurations in comparison to the balanced one for the strong consistency level. The configuration with equally–divided pools results in a huge energy waste that exceeds 54% and performance drop of 55%. Similarly, the mostly cold configuration causes an energy waste of roughly 50% and performance drop of 51%. Such losses are caused primarily, by the low number of warm nodes (half nodes for the first and only a third for the latter), which results in a performance drop thus, increasing the execution time. On the other hand, the mostly warm configuration results in approximately 27% of energy loss and 28% of performance drop. This demonstrates that for this type of workloads, the best configuration when consistency is mostly strong is a balanced configuration.

From these results, we conclude that a self–adaptive reconfiguration is necessary to reduce the energy consumption related to storage. The self–adaptive approach must consider the observed throughput of the storage system and the most used consistency level in the workload. Accordingly, the sizes of the warm pool and the cold pool are computed.

6.4 Discussion

Multiple analysis studies related to consistency were conducted over the years [126, 17, 29]. *Wada et al.* [126] investigate the level of consistency provided by the commercial cloud storage platforms. Accordingly, they analyze the correlation between the consistency provided and both the performance and the cost. In [17], the authors study past workload executions in order to verify consistency properties and the level of guarantees provided by the underlying key/value store. In a similar approach, *Bermbach et al.* study consistency properties within Amazon S3. The goal of this study is to investigate how old is stale data served within S3 cloud storage.

Energy consumption in the datacenter is an issue of extremely high importance. In this context, few approaches that attempt to reduce energy consumption for storage systems (underlying file systems for Hadoop mostly) were proposed [81, 16, 86]. GreenHDFS [81] is an energy-conserving variant of HDFS [69]. GreenHDFS divides the Hadoop cluster into *Hot* and *Cold* zones where a zone temperature is defined by its power usage as well as the performance requirements. Within GreenHDFS, data is classified in order to be placed in either zone. The classification aim is to enlarge the idle time of servers within the cold zones by assigning to them the least solicited data. Another technique that was proposed in or-

der to reduce the energy consumption of a cluster–based file system is Rabbit [16]. Rabbit introduces an efficient data layout as to provide ideal power-proportionality down to very low minimum number of powered-up nodes. Therefore, during periods of low utilization, a subset of nodes in the storage cluster can be powered down in order to save power. In a different approach, *Lang et Patel* propose a technique for managing Hadoop clusters, named the *All-In Strategy (AIS)* that consists in using all the cluster nodes to serve the workload. Once the workload execution comes to an end, all the cluster nodes must be powered down. The authors examine the *Covering Set (CS)* techniques that rely only on a subset of nodes in the storage cluster to serve workloads while powering down the others and conclude that AIS achieves higher energy savings than CS in most cases.

The energy consumption in the datacenter is becoming a pressing issue with many studies emphasizing its importance [55, 47]. However, few studies were dedicated to investigate power and energy management within storage systems (that were mostly dedicated to Hadoop). Moreover, no study –to our knowledge– addressed the impact of consistency management on the energy consumption. In contrast to related work, we introduce a first study that analyzes and shows how consistency can affect the energy consumption of the storage system. Furthermore, we show how a simple practice such as adapting the configuration of the storage cluster according to the consistency model can achieve significant energy savings.

6.5 Summary

In the era of Big Data and with the continuous growth of the datacenter scale, energy consumption has become a pressing factor in recent years. Similarly, consistency management has become of even higher importance for storage systems that operate at massive scales. In this study, we have highlighted, for the first time, the impact of consistency on energy consumption in the datacenter. Therefore, we have shown by means of experimental evaluation how the choice of consistency level affects the energy consumption of the storage cluster. We have demonstrated that the energy consumption is much higher with strong consistency levels. In contrast, the weakest consistency levels reduce, significantly, the energy consumption but at the cost of high rates of inconsistency. Quorum–based levels are middle ground consistency levels that save a reasonable amount of energy without tolerating stale reads. We conclude that –when update conflicts are efficiently handled– the basic eventual consistency is the best choice to save energy with just a small fraction of stale reads under light accesses while quorum–based levels are a better choice under heavy accesses. In addition, in our analysis, we have demonstrated the presence of bias in the storage cluster with eventual consistency levels. Thereafter, we have introduced a cluster reconfiguration into warm and cold pools as an adaptive solution to further save energy with eventual consistency. Our experimental evaluation has shown that such a solution leads to energy–saving, enhanced performance, and a reduced the stale reads rate.

Chapter 7

Chameleon: Customized Application-Specific Consistency by means of Behavior Modeling

Contents

This work started in the context of a 2-month internship at the Polytechnic University of Madrid (UPM)

In Chapters 4, 5, and 6, we have studied consistency management and its related trade-offs in cloud storage systems. Accordingly, we have proposed self-adaptive solutions

that handle consistency efficiently providing consistency when necessary while enhancing the storage system performance, reducing its monetary cost, and consuming energy in a more efficient way. However, these solutions are focused on the system side while having little consideration for the application (only read and write rates for *Harmony* and *Bismar*) and lacking the application semantics —except for *Harmony* that takes a small hint on the application consistency requirements, that should be specified by the application administrator, as an input—. In this chapter, we focus on the application level as a complementary work to our solutions at system level. The main target is to provide a big enclosing picture of customized and automatized consistency management that considers all influencing factors such as performance, monetary cost, and energy consumption. Therefore, in this work, *Chameleon* learns about the application behavior, captures its consistency requirements, and efficiently handles consistency tradeoffs on the system side when necessary. The resulting consistency management is exclusively specific to the application.

7.1 Motivation

Applications are different and so are their consistency requirements. A web shop application for instance requires a stronger consistency. Reading stale data could, in many cases, lead to serious consequences and a probable loss of client trust and/or money. On the other hand, a social network application requires a less strict consistency as reading stale data has less disastrous consequences. Understanding such requirements only at the level of the storage system is not possible. Both applications may impact the system state in the same manner at some points in time, for instance during holidays or high–sale seasons for the web shop and during important events for the social media. However, and while observing the same or similar storage system state, the consistency requirements can be completely different depending on the application semantics. In this work, and in contrast to related work, we focus on the application level with an aim at full automation. We argue that in order to fully understand the applications and their consistency requirements, such a step is necessary. Moreover, automation is of extremely high importance given the large scales and the tremendous data volumes dealt with within today's applications [122].

Many adaptive consistency models were proposed over the years such as our approach *Harmony* in Chapter 4, and consistency rationing [82] (presented in Chapter 3). These approaches were mainly proposed in the purpose of dealing with dynamic workloads at the system level. They rely, commonly, on the monitoring data of the accesses in the storage system. Moreover, *Harmony* requires a small hint about consistency requirements (while consistency rationing focuses more on the cost of consistency violations). In contrast, we introduce *Chamelon* for a broader context —considering a wider range of workloads and consistency policies— to operate at the application level –instead of the system–. *Chameleon* therefore, identifies the behavior of the application (and subsequently identify when workloads exhibit dynamicity) in order to efficiently manage consistency. Moreover, consistency requirements must be understood at the application level. Thereafter, when a dynamic policy such as *Harmony* is needed (for a given time period), the application tolerable stale rate can be computed automatically and communicated to the consistency policy (*Harmony*).

Chameleon, based on machine learning techniques, models the application and subsequently provides customized consistency specific to that application. The modeling is an

offline process that consists in several steps. First, multiple predefined access pattern metrics are collected based on application past data access traces. These metrics are collected per time period. The chronological succession of time periods presents the application *timeline*. This timeline is further processed by machine learning techniques in order to identify the different states and states transitions of the application during its lifetime. Each state is then automatically associated with an adequate consistency policy from a broad class of consistency policies. The association is performed according to a set of specified input rules that reflect precisely the application semantics. Thereafter, based on the offline built model, we propose an online prediction algorithm that gives a prediction on the state and the associated consistency policy for the next time period of the application.

7.2 General Design

7.2.1 Design Goals

In order to build our approach, we set three major design goals that *Chameleon* must satisfy:

Automated application behavior understanding. In order to fully capture the consistency requirements of an application, it is very important to learn about its access behavior. The behavior of an application is responsible, in no small part, of defining the consistency needs. In general, the behavior of an application (such as Web services) changes over time periods and thus, expresses a degree of load variability. In this context, the behavior modeling of an application requires automation as it is extremely difficult for humans to perform such a task given the scales of today's Big Data applications and the tremendous amount of information to process during their life cycles . Therefore, our target is to provide the two following automatized features: *a robust behavior modeling* of applications, and *an online application behavior recognition*.

Application semantics consideration. Understanding the application behavior is necessary but not sufficient in order to behold the consistency requirements. The high-level semantics of what the application need can only be provided by humans. For instance, defining what are the situations for which reading stale data is harmless or indicating whether update conflicts are managed (within the application or the storage system). In this context, these information are critical in order to provide the desirable level of consistency. Therefore, these application semantics should be provided in the form of rules that help associate every specific behavior of the application with a consistency policy. Hereafter, we propose mechanisms that make the task of rules setting and consideration simple and efficient.

Customized consistency specific to the application. A wide range of Big Data applications are in production nowadays. These applications, generally, operate on very wide scales and have different behaviors as well as different consistency requirements. Therefore, our goal is to exploit these differences by examining the application behavior in order to provide adequate consistency policy that is fully specific to the application. Such customized consistency allows the application to fulfill its target by providing the level of consistency required in every time period while optimizing SLA objectives such as performance, availability, economical cost, and energy consumption whenever it

is possible. In this context, the application selects the most appropriate consistency policy whenever its current behavior changes.

7.2.2 Use Cases

Hereafter, we show few applications that express, in general, access behavior variability (which is considered one of the main challenges of Big Data by SAS [31] as it was shown in Chapter 2). In this context, we show how these applications can, potentially, benefit from customized consistency.

Web Shop. A typical web shop that provides highly available services worldwide exhibits high load variability. During the busy holidays periods (*e.g.* christmas holidays), the underlying distributed storage or the database observes extremely heavy data accesses loads as the sales grow very high. Similarly, high loads are expected during discount periods. In contrast, the periods just after high–sale seasons are usually very slow resulting in very small number of data accesses at the storage system level. The system can experience an average load outside the aforementioned periods. Moreover, the data accesses may exhibit different patterns considering the level of contention to keys, the number of successive reads, the waiting time between two purchases of the same item etc. Our customized consistency approach, *Chameleon*, is designed for this purpose of access pattern (state) recognition in order to select the most pertinent consistency policy that satisfies the application requirements avoiding undesirable forms of inconsistency. A web shop' one–year lifetime can be divided into 52 week-based time periods. Every time period exhibits an application state and is assigned a consistency policy specific to that state.

Social Media. Social networks are typically worldwide services. Similar to Web shops, social media can exhibit high levels of load variability. High loads are expected during important events and breaking news (for general-purpose social networks for instance) while medium and low loads can be perceived when regular or no events are trending. Moreover, the load variability is location oriented. For instance, high loads –associated with a big event– that are perceived in one country will not be necessarily perceived in other locations. Therefore, data access patterns may vary in their loads but also in their behavior considering contention, locality and so many other criteria. Customized consistency can be applied for this type of applications to capture the states of the application in order to recognize access patterns and locality-based behavior. Accordingly, *Chameleon* selects adequately the consistency policy to adopt for a given time period. The application lifetime in this case can be a year long and a time period length equals one day for instance.

Wikipedia. Wikipedia is a collaborative editing worldwide–available service. While Wikipedia load variability is not comparable to the one of a web shop, it can still affect the consistency requirements. Moreover, the contention of accesses to the same keys is an important factor. Most of the time, contention is very low, mainly because of the large number of articles within Wikipedia covering a wide range of thematics. However, contention may change abruptly with articles becoming suddenly popular regarding their relation with recent breaking news or actuality events. Moreover, Wikipedia is a locality-based service.

It is only normal that people would like to access articles written in their local languages (or the English language). Customized consistency can therefore, recognize the data access pattern in order to select the consistency policy considering Wikipedia tolerance for staleness. Staleness can be tolerated to a certain degree as reading a stale article means missing the last update to it, which is, in most times, not very important and consists of a line or small information addition. Therefore, Wikipedia service administrator can just specify an inconsistency window that should not be violated in order to preserve the desired quality of service. For instance, setting this window to one hour can be fairly acceptable as Wikipedia is not a news feed. In this case, when a client reads an article it is guaranteed that it encapsulates all the updates that have been issued more than one hour ago.

7.2.3 Application Data Access Behavior Modeling

The first step towards customized consistency is to build a model that expresses different states and patterns of the application in order to facilitate the task of apprehending its consistency requirements.

7.2.3.1 General Methodology

First, we introduce the global view of our methodology as to build an efficient application behavior modeling approach that must be automated with a high efficiency. The model will be leveraged in the next step as to provide insight into future consistency requirements of the application. The modeling phase starts with identifying different application states. Moreover, each state is associated with a consistency policy that is best fit. In addition to state identification, the model must specify the state transitions of the application in order to reflect its lifetime behavior. The behavior modeling process requires the provision of past data access traces of an application as well as some pre-defined rules in advance.

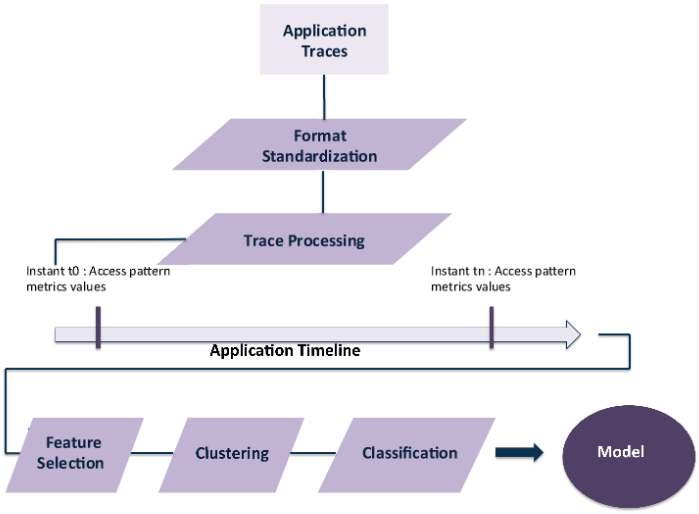
In the following, the different steps of the offline model construction, as shown in Figure 7.1, are presented. These steps are fully automated.

Application timeline construction. In this phase, application data access traces are processed in order to retrieve and compute a set of data access pattern metrics. These metrics are computed by specific time periods. The output of this phase is the application timeline where each time period of the timeline consists of a set of metric values. The application timeline refers to an interval of time that characterizes the application.

Application states identification. After the construction of the application timeline, all the time periods are processed using clustering technique as to identify the different states exhibited by the application.

States classification. After the identification of the application states, an offline learning process is performed. It consists in training a classifier based on the output dataset of the clustering technique. As a result of this phase, an online classifier that can recognize the application state or pattern at a given time period is built.

Consistency-State association. In this part, an algorithm that associates with each application state a consistency policy is introduced. This algorithm relies on defined rules and



First the timeline is constructed; then all time periods in the timeline are processed by unsupervised learning mechanisms for states identification; after states has been identified, states classification is performed.

Figure 7.1: Application behavior modeling

application-specific parameters that reflect high level insight into the application semantics. Consistency policies include strong and eventual consistency policies, static and dynamic policies, and local and geographical policies.

The resulting model can be represented as a directed graph $G(V, E)$ as shown in Figure 7.2 where vertices in V represent states and the edges in E represent the transitions that are chronological time successions as they are observed in the application timeline. The choice of a directed graph as a data structure facilitates the model storage providing efficient accesses to its different components. The computer representation of our model consists of an object that encapsulates the four following components:

The online classifier. This is the result of the states classification phase. This classifier is able to recognize the access pattern (the state) of the application for a given set of observed attributes (metric values).

State–Probability map. This map contains for each state identified its probability computed straightforwardly in the form of the occurrence number of the state by the number of occurrences of all states (number of time periods) fraction.

State–Policy map. This map is computed by the state-policy association algorithm.

Transition set. This set contains all the transitions between states (as defined by both the application timeline and states identification) with their respective ranks. The rank of a transition is its appearance order in the application model as it can be deduced

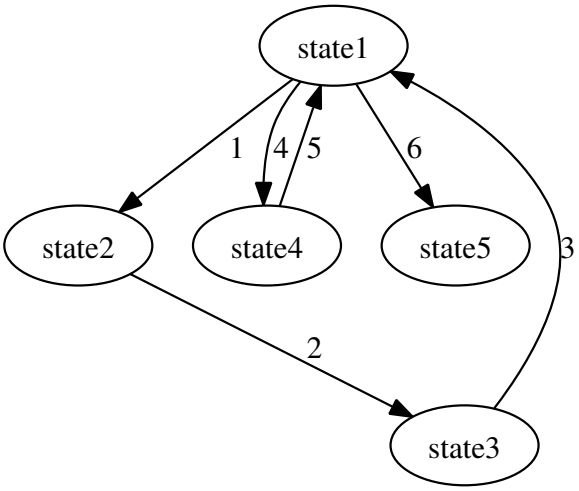


Figure 7.2: Application model example represented by a directed graph $G(V, E)$

Table 7.1: Access pattern metrics

Metric	**Value**
read_rate	Number of reads divided by the length of the time period
write_rate	Number of writes divided by the length of the time period
contention	Average contention to one key
no_possible_stale	Number of reads that may be potentially stale (depending on staleness rule)
min_successive_reads	Minimal number of successive reads to the same key
min_successive_writes	Minimal number of successive writes to the same key
avg_successive_reads	Average number of successive reads to the same key
avg_successive_writess	Average number of successive writes to the same key
max_successive_reads	Maximal number of successive reads to the same key
max_successive_writes	Maximal number of successive writes to the same key
avg_wait	Average waiting time between two operations
std_dev_wait	Standard deviation of the waiting time between two operations
min_wait_2_writes	Minimal waiting time between two writes to one key
avg_wait_2_writes	Average waiting time found between two writes to the same key
local_access	Local access from the same location or remote access

from both the chronological succession of time periods in the timeline and the results of states identification. An additional transition from the state of the last time period to the first is added with the highest rank.

7.2.3.2 Timeline Construction: Traces Processing

The application traces are converted to a standard format in order to be processed by our approach. The traces are organized by time periods and each time period is stored in a specific file. Within every time period, metrics that exhibit the access pattern are extracted. These metrics are shown in Table 7.1 and express the behavior of the application when accessing data within the storage system.

These metrics characterize the access pattern and therefore, their values enable the capturing of the application state. This in turn, provides an insight into the consistency requirements. These metrics values are determinant to the states characterization, but to the consistency policy-state association as well. Metrics, such as the read rate, the write rate, and contention can help define the level of consistency strength needed. Similarly, the mean waiting time and the standard deviation of waiting time between two operations express the degree of dynamicity and variation exhibited by the application. On the other end, the minimal waiting time between two updates provides hints on the possibility of update conflicts.

At the end of this phase, the application *timeline* is built. It is composed of the chronological succession of time periods where every time period has specific access pattern metric values.

7.2.3.3 Identification of Application States

States identification is one of the critical tasks in the model building process. In order to achieve efficient states identification at large scale with full automation, we rely on machine learning techniques that are able to process large datasets with relatively large number of attributes. In this context, we construct a dataset from the application timeline. Every instance represents a time period identified by a unique key, which is the timestamp of the period start. The access pattern metrics represent in this case the instance attributes. Our total number of attributes is 15. However, not all of these attributes may be relevant for all types of applications and therefore might affect badly the quality of both the supervised and unsupervised learning (ie. clustering and classification). In this context, we apply a feature selection algorithm that ranks the attributes in order of their relevance. Consequently, the unsupervised learning in the form of clustering is applied on the dataset with exclusively relevant features in order to identify the application different states.

Feature Selection. In machine learning, sample data is processed in order to extract knowledge that can be used for multiple aims. One of the most common objectives is to use this knowledge in the purpose of making predictions about future data. In this context, the quality of the data sample used for training is critical to reach efficiency with learning algorithms. Feature selection refers to the algorithms family that given the set of attributes selects a subset of relevant features and eliminates the irrelevant and redundant data that may make the knowledge discovery more difficult and cause overfitting problems [132].

In order to enhance the accuracy of our prediction model and avoid overfitting, we use *Recursive Feature Elimination* (*RFE*) [68, 41]. *RFE* best–fit use cases include small and average datasets with high dimensionality. Therefore, *RFE* is ideal for our model considering our sample dimension and that the number of time periods in the application timeline is limited. *RFE* is, commonly, used with Support Vector Machines (SVM) and was originally applied to gene selection for cancer classification [68]. It helped scientists to discover novel information on genetics that was constantly missed in the past when using other techniques.

Data Clustering. In this context, clustering is applied to automatically identify the application states. Multiple clustering algorithms exist. However, for our model we seek an algorithm that requires minimal information from the user. In particular, we have no assump-

tions on the number of states (clusters), or their shapes that may be completely arbitrary. As a result, common algorithms such as k-means [93] or Gaussian Mixture EM clustering [51] show limitations over their applicability to our specific problem. Both algorithms can find only linearly separable clusters while the first one requires the number of clusters in advance. However, at this stage we have minimal knowledge on the model and we do not put any assumptions during its construction. One clustering algorithm that fits our requirements is *density-based spatial clustering of applications with noise DBSCAN* algorithm [56]. We choose *DBSCAN* since it does not require the number of clusters but infers it implicitly in the computation. Moreover, it can detect arbitrarily shaped clusters.

DBSCAN is a density-based algorithm. Clusters are formed based on density reachability. Therefore, density variations are detected in order to form separated clusters of points. All points in the same cluster are mutually density reachable directly or by transitivity. *DBSCAN* requires two parameters, the first one is the distance used to define density reachability, and the second one is the minimum points that a cluster should encapsulate. Defining the two parameters is known to be a difficult task in DBSCAN. In our model, the number of minimum points in a cluster is one since one time period in the application timeline may exhibit a state that is different from all other time periods. In contrast, defining a precise value of the distance parameter is more difficult. We plan to propose a heuristic that computes a near optimal value of this parameter as a future work.

7.2.3.4 State Classification

The next step in our model is to build an automated online classifier of application states. In order to provide such a feature, we use supervised machine learning techniques. The classifier utilizes a learning model that collects knowledge from an input sample offline in order to provide data analysis and patterns recognition abilities that will be used for instance classification. Based on the states identified in the previous step, a learning sample will be passed to the classifier. The learning sample consists of a set of time periods data, and for each time period its state. Multiple types of classification techniques have been introduced. Two of the most efficient ones are Neural Networks [94] and Support Vectors Machines [33, 46]. Support Vector Machines (SVM) learning models were first introduced in 1992, and gained major popularity since. They are linear classifiers and outperform most of statistical and learning techniques such as Neural Networks. In our model, we choose an SVM algorithm mainly because of its efficiency with high dimensionality data. Unlike Neural Networks, SVM techniques perform well with large datasets providing the necessary means to analyze data and find separations efficiently at high dimensionality. SVM represents data in a set of multidimensional space, then constructs a set of hyperplanes based on vectors in order to separate data points.

7.2.4 Rule-based Consistency-State Association

The process of identifying the application states lacks the consistency context. Therefore, states are identified based on access patterns that should be associated with the pertinent consistency policy based on the application semantics. In this part, we propose an algorithm that performs such an association.

7.2.4.1 Consistency Policies

Multiple consistency models were proposed over the years. Cloud storage systems usually implement a unique and well defined consistency model per system. However in recent years, as introduced in Chapter 3, multiple distributed storage systems that implement various models in order to provide different levels of consistency were introduced. Moreover, these systems provide flexible APIs to the client in order to select the consistency level on per operation basis.

In this work, we categorize a set of consistency policies considering the following properties:

- Strong policies vs. Eventual policies
- Static policies vs. Dynamic policies
- Local policies vs. Geographical policies

In this context, the following consistency policies are considered by the State–Policy association algorithm:

Strong Consistency. This is the *static strong* level of consistency by means of synchronous replication involving all the replicas in read and write operations in order to avoid any form of inconsistency.

Eventual One Consistency. This is the basic level of *static eventual* consistency that involves only one replica —mainly the fastest replica to respond— in read and write operations.

Eventual Quorum Consistency. This policy is a *static eventual* policy. The number of replicas involved in the access operations is equal to $\lfloor (\frac{number\ of\ replicas}{2}) + 1 \rfloor$. Therefore, the fastest replicas to respond are considered in this policy no matter their locations (*e.g.* which datacenter, which rack etc.).

Eventual Local_Quorum Consistency. In this *static eventual local* policy, only a quorum of replicas local to the accessed datacenter are considered in the read and write operations. Remote replicas (in other data centers) are not solicited in order to avoid high network latencies between data centers.

Eventual Each_Quorum Consistency. In contrast to the previous one, this *static eventual geographical* policy considers local quorums in every datacenter for read and write operations.

Dynamic Performance Consistency. In this policy, the goal is to *dynamically* tune the consistency level for dynamic workloads with a particular focus on improving performance when possible. In our model, we use our approach *Harmony*, introduced in Chapter 4, as the *Dynamic Performance Consistency* policy. Subsequently, the tolerated stale read rate of an application is computed automatically. Its value is equal to $100 - \frac{undesirable\ stale\ reads}{number\ of\ reads} \times 100$ where the undesirable stale reads can be computed from the traces based on the input rules as it will be explained in the next section.

Dynamic Cost Consistency. This is a *dynamic policy* that adaptively tunes the consistency level as well, with the goal of reducing the monetary cost of using the infrastructure (mainly cloud platforms) when possible. In our model, *Bismar*, introduced in Chapter 5, is the *Dynamic Cost Consistency*.

Dynamic Energy Consistency. In this *dynamic* policy, the adaptive selection of the consistency level must take into account the impact on energy consumption. In the absence of such a policy at the current time, we can either use *Harmony* or simply rely on the storage system reconfiguration at runtime, as presented in Chapter 6, in order to reduce the consumption. As part of future work, we plan on implementing a consistency policy where the main focus is to save energy when possible.

7.2.4.2 Input Rules

In order to reach a meaningful state-policy association, input information is required to be provided by the application administrator. Moreover, the administrator has the choice to either select one of the generic rules included in our model or implement its own specific rule based on its application semantics. These rules are used in the model to flag situations when staleness is not desired and may be harmful. Our model is provided with mainly two generic rules. The first rule is named *the stock rule*. It defines the number of the "consuming" updates to the same key per time period before flagging the successive reads as potentially reading undesirable values. This rule is based on the example of stock variables in web shop applications. As long as the value of such a variable is higher than a threshold value, the exact value of the variable is not required in order to avoid product availability conflicts that lead to anomalies. Therefore, stale data in this case is acceptable as it will not lead to the purchase of unavailable products. The second rule, surnamed *the inconsistency window rule*, is based on a provided inconsistency window value. The value of the inconsistency window is used later to compute the number of potential stale reads. In addition, for systems that are more tolerable for staleness, a second inconsistency window specific to the application is introduced. Its value is the maximum staleness interval that should not be exceeded. For instance, let consider the example of a social network where a given person friends should be able to read his updates that were committed more than given period of time ago (*e.g.* two minutes ago). The stale reads that exceed this value are considered as undesirable (since they affect the quality of service for this case for instance). These two rules are provided in the following XML format:

```
<rules>
    <allowed_staleness>
        <type> stock or icw </type>
        <stock> <min> 50 </min> </stock> <!-- or -->
        <inconsistency_window>
            <native> storage_incon_window <native>
            <tolerated> application_incon_window <tolerated>
        </inconsistency_window> <!-- or -->
        <flag> no or yes </flag>
    </allowed_staleness>
</rules>
```

Additional rules are necessary as well. A rule that specifies whether the update conflicts management is provided in the storage configuration or not is expected. If conflicts are not handled, any potential conflict situation would automatically lead to the application of the strong consistency policy. Moreover, the application administrator is required to specify whether dynamic policies are allowed to be selected (if the specific conditions are met) or not. When dynamic policies are allowed, a variation threshold and the focus of the dynamic policy are required. The variation threshold will be compared to the standard deviation of waiting times and therefore, can be specified as $\alpha \times$ *mean waiting time* where $0 < \alpha < 1$. These two rules are specified in the following XML format:

```
<rules>
    <conflict_handling>
        <flag> yes or no </flag>
    </conflict_handling>

    <allow_dynamic>
        <flag> yes or no </flag>
        <variation_threshold> value </variation_threshold>
        <policy> performance or cost or energy </policy>
    </allow_dynamic>
</rules>
```

7.2.4.3 State-Consistency Association Algorithm

Figure 7.3 shows the state-policy association algorithm in a decision tree-like form. This is an offline algorithm. It takes as input the model with its identified states, the set of consistency policies, and the XML file with the input parameters and rules. For every state, the association algorithm aggregates metrics values from time periods that belong to that state. The algorithm starts by checking both the existence of update conflicts and mechanisms to handle them in order to exclude or select the *strong* consistency policy. In the absence of conflicts, eventual quorum-based consistency levels can guarantee strong form of consistency. Therefore, the following step is to check the number of potential stale reads based on the provided staleness rule. Accordingly, either the basic eventual consistency level that involves one replica is selected or further processing is required. In the latter case, the algorithm checks whether staleness is allowed for the application. Subsequently, it selects *Quorum-based* levels if staleness is not allowed. The *Local_Quorum* policy is chosen when accesses are potentially all or in–most directed to their local data centers. In contrast, the *Each_Quorum* policy is chosen for the non-local accesses (accesses to different data centers). On the other hand, if staleness is allowed, the algorithm checks whether dynamic policies are allowed as well. Hereafter, the dynamicity of the application workload is investigated based on the observed standard deviation of waiting times between data access operations. According to the variation threshold, the dynamic policy might be applied based on the desired optimization focus (*ie.* performance, cost, or energy consumption). In the case where dynamic policies are not allowed (by the application administrator), a default policy is selected. Since in this case, staleness is allowed, we therefore choose the basic consistency level *One* as a default policy.

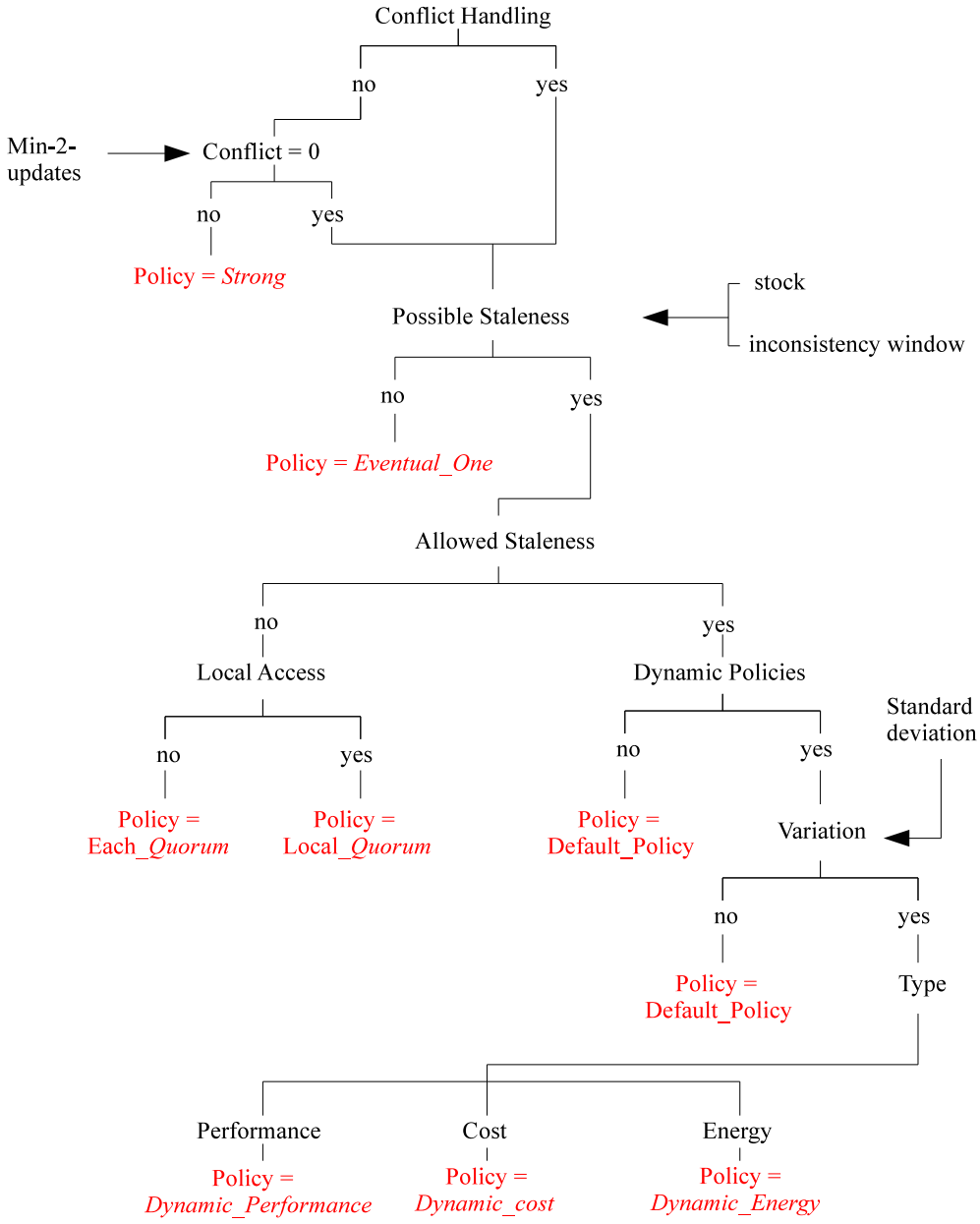


Figure 7.3: State-Consistency Association Algorithm

7.2.5 Prediction-Based Customized Consistency

Algorithm 3: Next State Prediction

```
Input: Model, stats
Output: State
currentState ← Model.Classifier(stats)
if currentState successors set size = 0 then
    nextState ← defaultState
else
    if currentState successors set size = 1 then
        nextState ← currentState successor
    else
        nextState ←
        recPrediction(currentState, predecessor(currentState), successors(currentState))
    end
end
Function recPrediction(state1, state2, successors)
    Max ← random state in successors
    tr ← Transition(state2, state1) rank
    for state in successors do
        if state rank < tr then
            successors ← successors − {state}
        if State Probability > Max Probability then
            Max ← State
    end
    if successors set size > 1 and predecessor(state2) != ∅ then
        return recPrediction(state2, predecessor(state2), successors)
    else
        if successors set size = 0 or predecessor(state2) = ∅ then
            return Max
        else
            return the successor in successors
        end
    end
end
```

After the offline model construction has been completed, we leverage the model as to provide customized consistency for the running application. Henceforth, we provide an online algorithm that at the end of every time period, and based on the model, gives a prediction on the expected state that the application should exhibit for the next time period. Accordingly, the consistency policy to adopt is chosen by a simple state-policy mapping. The prediction mechanisms are shown in Algorithm 3. The algorithm requires the application model and the current observed access pattern metrics (referred to as stats) collected from the application logs. The model consists of 4 components: the *Online state classifier*, the *State–Probability map*, the *State–Policy map*, and the *Transition set*. The algorithm starts by classifying the current state of the application. Once the current state determined by the online classifier, its successor set in the application model is computed based on the transition set.

In the case where this set is empty, the state with the highest probability is selected. If on the other hand, the set contains more than one successor, a recursive function *recPrediction* is called. This function leverages the model in order to compute which state the application is expected to transform to.

The recursive function *recPrediction* checks the transitions of the application to the current state starting by the recent transition to the least recent. In this context, the states in the successors set are either ruled out or conserved based on their ranks. Therefore, only states that are expected after the succession of the states exhibited so far are not ruled out. The recursive call traces the transitions from the recent to the old one until only one or no state belongs to the successors set. In the case where the successor set is not empty nor include only one element while the oldest transition has been reached, the state with the highest probability is selected.

7.3 Implementation and Experimental Evaluations

In this section, we first describe the implementation of *Chameleon*. Secondly, we present our experimental evaluation. The main goal is to demonstrate the quality of the modeling behavior approach and to show the efficiency of *Chameleon* in adapting to the need of the application at full automation, which is the goal we were seeking when designing it. In this context, the recognition of the application behavior and its requirements is performed by the online classifier (SVM), which is a black box to the human users.

7.3.1 Implementation

Chameleon targets a wide range of large-scale applications. These applications might be completely different. Moreover, their traces are potentially heterogeneous in both content and formats. In order to deal with such a heterogeneity, *Chameleon* is designed to easily integrate new data formats and to automatically deal with load variability. *Chameleon* implementation is divided into two main parts: the traces parsing part, and the modeling and the online prediction part.

Traces parsing and timeline construction. At the start of this phase, the application traces (logs) are converted into a standard format. We define the standard format where each line consists of the following values separated by one blank: *timestamp of the operation*, *the key of accessed data*, *the type of access operation*, and, if possible to get this information, *a flag to indicate whether the geo–location of the client is local or not*. This approach makes the integration of any new application (and its traces) easy by just writing the corresponding converter to the standard format. The traces in the standard format are processed by a *Python* built module. This module divides the large traces log file into time periods based on the timestamps and then stores every time period in a separate file. Afterwards, every time period file is loaded at a time from the disk and then processed in order to efficiently use memory space. Hereafter, all computed metrics values of each time period are stored in one file that represents the application timeline.

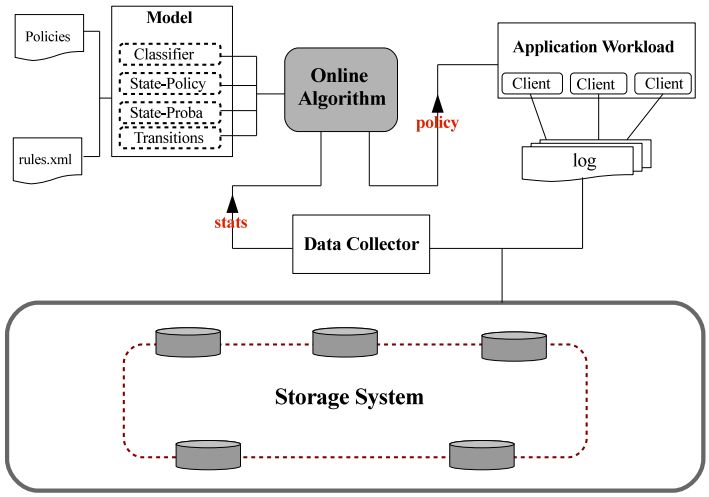


Figure 7.4: Overview of Chameleon Implementation: The online algorithm relies on the application model and the stats collected from client logs in order to select the adequate consistency policy for the next time period

Modeling and online prediction. The second part of *Chameleon* implementation builds the application model and the online prediction algorithm. This part is implemented in *Java* where the application model is an object. Figure 7.4 shows our implementation. The *model component* encapsulates the model data. The model is instantiated before runtime based on the application timeline, the input rules file, and the set of consistency policies. At runtime, every *application client* logs its access information. The *data collector* component collects the necessary data from client logs and extracts the data access statistics to be sent to the *online algorithm* component. The latter uses the knowledge acquired from the *model* component in order to perform the prediction computations of the next consistency policy to use during the next time period. The machine learning algorithms applied for state identification and classification were implemented based on the *Java Machine Learning Library (Java–ML)* [78]. *Java–ML* wraps over multiple other machine learning libraries in *Java*. The recursive feature elimination RFE implementation was Java–ML native while the clustering algorithm was wrapped over the DBSCAN implementation in *Weka* library [131] and the classification algorithm was wrapped over SVM in *LIBSVM* library [88].

7.3.2 Model Evaluation: Clustering and Classification

Wikipedia Traces. In order to validate our modeling phase, we investigated available traces of large-scale applications that fit our case study. A perfect use-case match is Wikipedia [135]. Based on access traces collection and analysis of Wikipedia described in [123], we use a traces sample collected in 1st of January 2008. The traces consisted of roughly 9056528 operations and a total log file size of approximately 1 GB. The access traces consisted of the timestamp of the operation occurrence, the page that was accessed, and the type of the operation. Unfortunately, the information related to the client geographical loca-

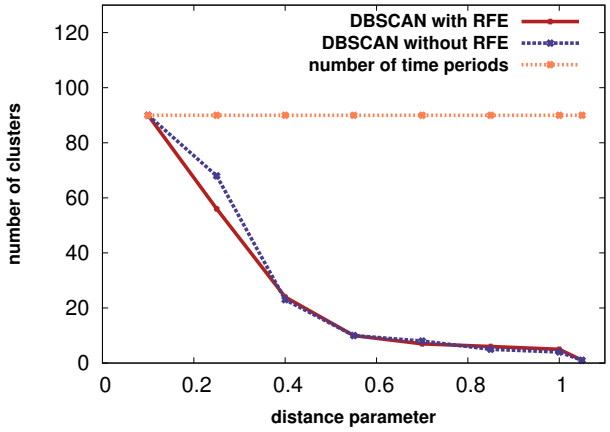


Figure 7.5: Clustering of time periods with and without RFE

tion was not available. These traces were processed to extract relevant metrics (as described in the modeling section) and construct the application timeline in order to perform experimental evaluations. For the purpose of experimental evaluation, the length of a time period is set to 100 seconds and the application timeline consists of 90 time periods.

Setup. The experimental sets to evaluate the offline modeling phase were conducted on a Mac equipped with Intel Core-2-Duo CPU of 2,66 GHz frequency, memory size of 4 GB and a 300 GB hard drive. The used operating system is Mac OS X 10.6. The *Java Runtime Environment* version was 1.6.

Clustering Evaluation. In order to validate our model, we ran a set of experiments oriented to evaluate the quality of the clustering algorithm DBSCAN used for states identification as well as the quality of the classification with SVM.

We first start by analyzing the DBSCAN algorithm and its behavior when modifying the input parameters, in particular the distance parameter since the minimal number of points that a cluster should enclose has been fixed to one (as one application time period can exhibit a behavior different from all others). Moreover, we compare the numbers of found clusters every time with and without recursive feature elimination RFE. Figure 7.5 shows the varying number of the formed clusters when varying the distance parameter in the interval 0.10 to 1.05 for the Wikipedia sample timeline. The number of clusters increases when decreasing the value of the distance parameter. The two clustering approaches with and without RFE find a close number of clusters. This shows that the application of RFE preserves the intuitive clustering results of *DBSCAN*, with all the features in the dataset considered, while enhancing greatly the quality of classification as it will be demonstrated. With the very small distance parameter value of 0.10, the number of found clusters for both approaches equals that of the number of time periods (data instances). This makes the clustering ineffective and the classification phase more difficult. Similarly, with the value of 1.05, the found number of clusters is one for both approaches making the whole modeling process trivial. Moreover, We can observe that with small distance parameter values, the clustering approach that is not preceded with RFE forms more clusters. This is mainly, because of the recursive and

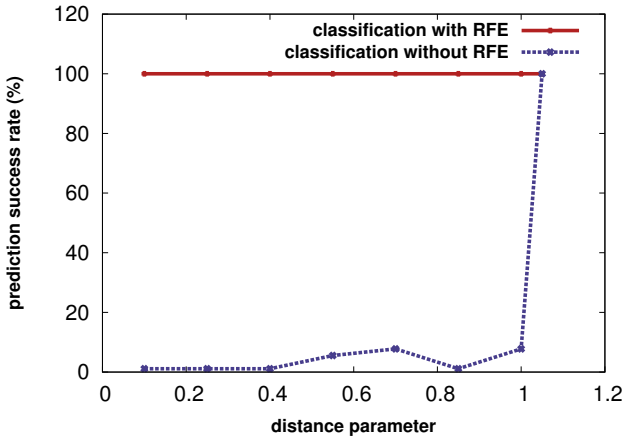


Figure 7.6: Accuracy of instance classification

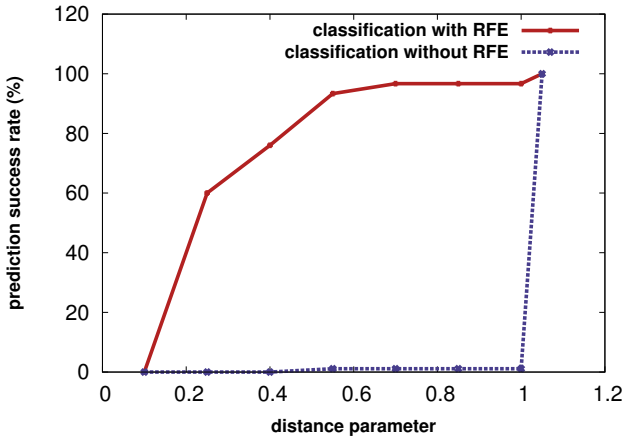


Figure 7.7: Classification of new data instances: accuracy

trivial features not being eliminated. In addition, these features (attributes) will affect badly the classification quality.

In order to analyze the impact of the two clustering approaches (with and without RFE) on the classification process, we train our SVM algorithm with the clustered output data of both approaches. After that, we classify every data instance in the Wikipedia timeline with the two classifiers (with and without applying RFE). Subsequently, we compare the predicted value by the classifier with the observed value from the clustering phase in order to determine the prediction success rate. Figure 7.6 clearly and conclusively shows the importance of the recursive feature elimination in the classification process. Clustering and classification without RFE results in a very mediocre classifier with very small fraction of successful predictions. In contrast, RFE makes the whole process very effective with 100% fraction of successful predictions on the training data. However, the classifier in this latter case might suffer from the overfitting to the data sample and may behave badly with outsider (new coming) data instances. Therefore, we conduct a set of experiments hereafter to evaluate the classification quality independently of the input dataset.

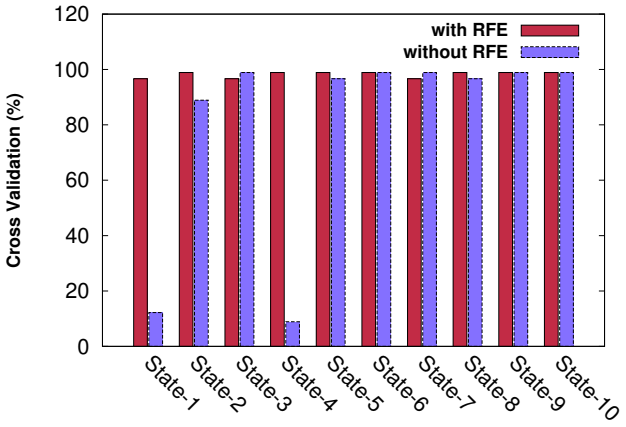


Figure 7.8: Cross–Validation of classification with different states

Cross–Validation. Recursive feature elimination RFE algorithm was applied mainly to enhance the quality of the classification. In order to evaluate the classification algorithm and to check whether it suffers from the overfitting problem, we ran a cross–validation evaluation. We divide the data resulting from the clustering phase into two sets. The first set that contains randomly–selected fraction of two thirds of data instances used to train the classifier. The second set contains the remaining data and is used for classification. Figure 7.7 shows the prediction success rate of both approaches with and without RFE where the classified data is different from the training data. As expected, the classifier without RFE performs extremely bad with very low, sometimes null, prediction success rates. In contrast, RFE–SVM classifier achieves very high success rates (over 96%) when the distance parameter value is greater than 0.55. Moreover, it performs reasonably well with relatively small distance values such as 0.25 (over 56% success rate). However, the classification proved ineffective with a distance of 0.10. The reason for such a behavior is the inefficiency of the clustering algorithm with this distance value since all data instances are classified in distinct classes. As part of a future plan, we intend to investigate the possibility of a heuristic that targets the computation of the distance value. Such a heuristic should aim at computing a value that provides high success rate while resulting in a number of clusters higher than a given number (for instance $\alpha \times \mathit{number\ of\ consistency\ policies}$).

Figure 7.8 shows the cross–validation for the different states of the application –observed in the data sample– when the distance parameter in the clustering phase equals 0.55. The cross–validation type used is *K-fold cross-validation* with k equals to 10. The 90 time periods (instances) were classified correctly for the big majority of states of the classification with RFE (Over 96% successfully classified for all states), which shows the robustness and the precision of our model that is data set–independent. In contrast, classification without RFE classifies instances at very low accuracy for few states and at high accuracy rates for others. However, the poor success rate shown in Figure 7.7 is caused essentially by the poor classification with states 1 and 4. In particular, because the number of time periods that exhibit these two states is significantly higher than others. Moreover, the successful cross-validation with other states is, mainly, because of true negatives rather than true positives.

7.3.3 Customized Consistency: Evaluation

Experimental Setup. We ran the experiment sets to evaluate the online model on Grid'5000 [79]. We used Cassandra as a hosting storage system. Therefore, we deployed Cassandra-1.2.3 version on two sites on Grid'5000 with 3 replicas in each site. 30 nodes were deployed on *Sophia* and 20 nodes on *Nancy*. Nodes in *Sophia* are equipped with 300 GB hard drives, 32 GB of memory, and 2-CPUs 8-cores Intel Xeon. The *Nancy* nodes are equipped with disks of 320 GB total size, 16 GB of memory, and 2-CPUs 8-cores Intel Xeon. In addition, the network connection between the two sites in the south and the north east of France is provided by *Renater* (The French national telecommunication network for technology, education, and research). At the time of running the experiment sets, the average round trip latency between the two sites was roughly 18.53 ms.

Micro–Benchmark. We designed a micro–benchmark that exhibits the same characteristics shown in the used Wikipedia traces. It was extremely difficult to reproduce exactly the same or closely similar behavior. The main challenge is that we do not have enough resources (with our 50–nodes Cassandra cluster and network speed) to reproduces the same extremely high throughput and small waiting times between operations exhibited in the traces. Therefore, we designed a micro-benchmark that keep Wikipedia load characteristics with amplified waiting times between operations. The designed workload is implemented in Java and divided into 90 time periods (the number of time periods computed from Wikipedia traces). Every time period consists of specific properties: read rate, write rate, contention to the same keys, and values of average waiting time between operations. These values are computed directly from the Wikipedia traces (with amplified waiting times) and saved into a configuration file. At runtime, the waiting time between operations is generated randomly following an exponential distribution where the mean equals the average waiting time. In order to run the online model evaluation, we first run the benchmark to generate new traces that will be used for offline model construction.

Online Prediction. In order to run *Chameleon*, we have selected the *inconsistency window rule* as an input rule to apply with Wikipedia. For the purpose of experimental evaluation and given our experiment scale, we assume that the tolerated stale reads are the ones that enclose updates older than one minute (60000 ms). Moreover, we consider that update conflicts are manageable since it is possible to solve them at a latter time by Cassandra based on causal ordering (Anti–Entropy operation). We consider in addition, that dynamic policies are allowed with a focus on performance. The variation threshold value is fixed at 25% of the mean waiting time.

Figure 7.9 shows the probability of observing the different states. We compare the probability of states observed from past access traces by means of clustering and the probability of predicted states at runtime. The results clearly show that for the states with the highest probabilities (mainly State-1 and state-6) the probability is closely similar, which demonstrates the efficiency of our online prediction model in predicting similar behavior to the one modeled. We can observe that for State-3 the probability at runtime is higher. This can be explained by many factors including the error margin that can be associated with classification, the random generation of waiting times in the workload, as well as the performance

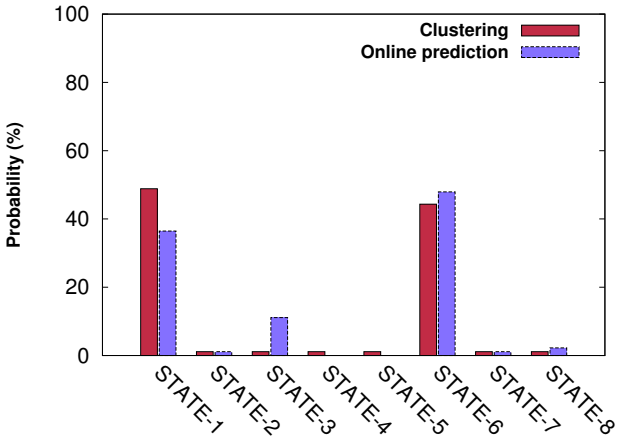


Figure 7.9: Observed states distribution

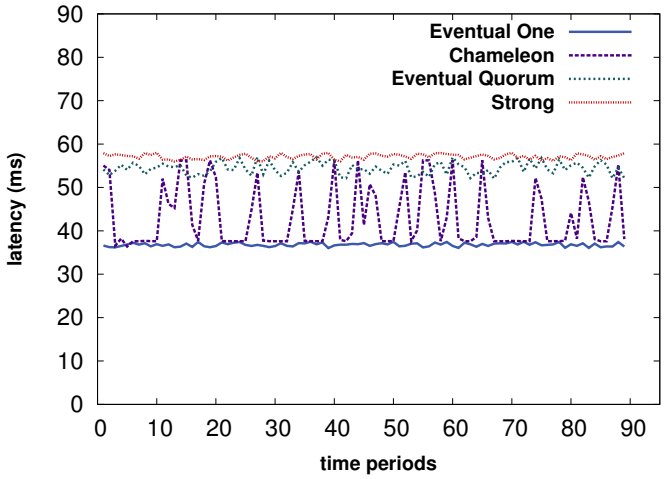


Figure 7.10: Latency evaluation

of the Cassandra storage system (that might be slightly affected in particular by events and traffic on the shared network) and might therefore, slightly affect the load characteristics.

Latency and Throughput. Figure 7.10 and Figure 7.11 show the performance exhibited by our approach *Chameleon* in comparison to different consistency policies. In Figure 7.10, we can observe that both the *Quorum* and *Strong* static consistency policies result in a high operation latencies throughout the different time periods of the workload. Such high latencies are mainly because these policies suffer from high wide-area network latency. In contrast, the eventual *One* policy achieves the lowest operation latencies since data are accessed from closest local replicas. However, with this policy, multiple consistency violations might occur. Our approach, *Chameleon*, shows a high latency variability across the different time periods while the latency is relatively low and close to the lowest one most of the time. This variability is the result of adapting the consistency policy according to the application behavior exhibited in that specific time period. Therefore, *Chameleon* provides specifically a consis-

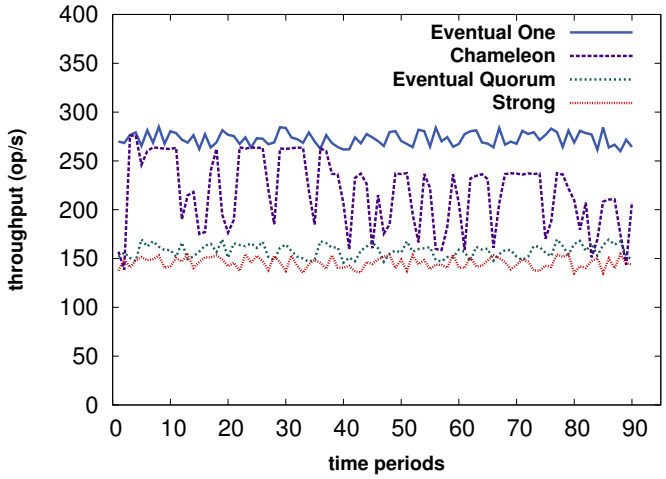


Figure 7.11: Throughput evaluation

tency policy in accordance to the application state for every time period.

Similarly, Figure 7.11 shows the throughput of the different consistency policies throughout the workload time periods. Both the *Quorum* and the *strong* policies show relatively low throughputs. The throughput in these cases suffers from the wide area network latency as well as the extra traffic in the network generated by these consistency policies. This can be in particular, penalizing for a highly on-demand service such as Wikipedia. The *One* policy on the other hand, exhibits much higher throughput but, at the cost of potentially consistency violations. *Chameleon* shows variable throughput (similar to latency) that depends on the behavior of the application in the given time period. Therefore, the consistency policy is selected to suit that behavior of the application. Although, a high level of variability is observed, *Chameleon* throughput is almost always better than the *Quorum*'s and relatively close to the throughput of the *One* policy.

Data Staleness. After the investigation of the different consistency policies, we show in Figure 7.12 the staleness of data throughout the workload timeline. The *Quorum* and the *Strong* policies provide high levels of consistency with no stale data being read. However, this comes at the cost of lower performance and availability, in particular if we consider that staleness is both rare (because of the low contention to data) and is accepted to a certain degree within Wikipedia. The *One* level allows the highest number of stale data, though these numbers are relatively small since the contention of data accesses to the same keys is not high. Moreover, the *One* policy might allow undesirable forms of staleness (*e.g.* reading very old data). *Chameleon* allows for very small number of reads to be stale. Moreover, these stale reads are expected in accordance with the specified rules during the state–consistency association phase. Therefore, tolerating such a small "unharmfull" fractions of stale reads comes with huge benefits for performance support, money savings, or energy consumption reduction.

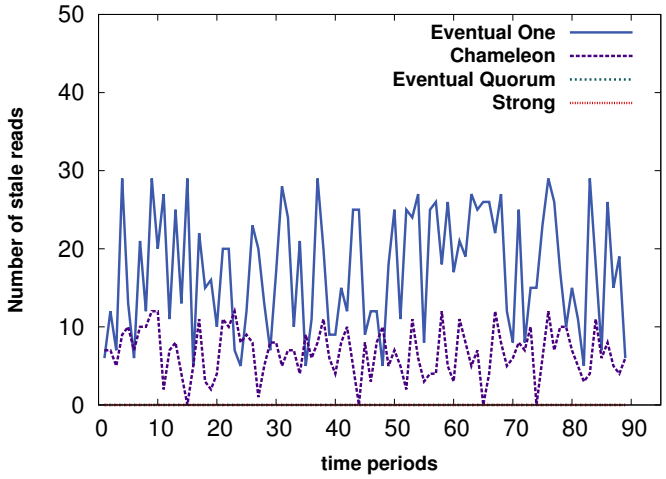


Figure 7.12: Data staleness evaluation

7.4 Discussion

Using past data access traces in the purpose of studying consistency was investigated in [17]. *Anderson et al.* proposed mechanisms that process past access traces offline in order to check few consistency guarantees and whether they were violated or not during runtime. The main purpose of this work is the analysis and the validation of the past execution of a workload in order to understand the provided consistency guarantees of a key–value store. One of their primary observations is that consistency violations are correlated with the increasing contention of operations to the same keys.

Few adaptive consistency models were proposed over the years [82, 128, 87, 90]. Most of these approaches focus on consistency management on the storage system level. Therefore, there is a little or no focus on consistency requirements on the applications side. Moreover, no extensive study of the application data access behavior was introduced. In [87], the authors propose a flexible consistency model that variates the level of guarantees depending on the nature of operation executed in relation with the application semantic. However, the selection of consistency level of one operation is not automatic and may be very difficult to induce by application developers, in particular with today' scales. *Kraska et al.* [82] propose an approach that imposes the consistency level on data instead of transactions. Therefore, data is divided into three categories depending on the consistency guarantees to be provided. The main difficulty with this approach is the lack of automation when categorizing the data, which may be in our case of Big Data applications a huge obstacle, since the administrator is required to categorize huge volumes of potentially heterogenous data manually. Both approaches in [128, 90] do not consider the application requirements as the first approach takes into consideration only the read rates and the write rates with no application semantics consideration while the second approach was proposed in the particular context of providing persistent storage to databases in the cloud.

Multiple works contributed to the area of modeling and characterizing large-scale infrastructures and applications. *Montes et al.* [97] proposed *GloBeM* a global behavior modeling for the whole grid. GloBeM rely on machine learning and knowledge discovery techniques

in order to represent the different sub-systems of the grid by one global model in the form of finite state machine. This model allows the system administrators to predict future grid state transitions and therefore anticipate any undesirable states. Other works focused on the characterization and analysis of large-scale application workloads and their behaviors. In [123], the authors study Wikipedia access traces and accordingly classify client requests and collect access metrics such as the number of read and save operations, and load variations. In a different work [28], the authors study the user behavior in online social networks. They propose a thorough analysis of social network workloads with regard to the user behavior in an attempt to enhance the interface design, provide better understanding of social interactions, and improve the design of content distribution systems. However, none of these studies were leveraged to provide a vision on consistency management.

7.5 Summary

Consistency management in large–scale storage systems has been widely studied. Most of existing work focused on leveraging consistency tradeoffs and the impacts on the storage system with a secondary minimal focus on the application consistency requirements. In this chapter, we focus instead on the applications in order to fully apprehend their consistency requirements. Therefore, we first introduce an automatized behavior modeling approach that detects the different states of an application lifecycle. These states are thereafter associated with adequate consistency policies based on the application semantics. The application model is leveraged in a second step as to provide a customized consistency specific to the application. At runtime, our approach, *Chameleon*, identifies the application state by means of efficient classification techniques. Accordingly it provides a prediction on the behavior of the application and subsequently selects the best-fit consistency policy for the following time period. Experimental evaluations have shown the efficiency of our modeling approach where the application states are identified and predicted with a success rate that exceeds 96%. Moreover, we have shown how our approach *Chameleon* is able to adapt specifically to the application behavior at every time period in order to provide the desired properties (including performance, cost, and energy consumption) with no undesirable consistency violations.

Part III

Conclusions and Perspectives

Chapter 8

Conclusions

Contents

We live in the era of Big Data. The recent explosion of data sizes and the related issues of velocity and variety introduced unprecedented challenges at enormous scales. Cloud Computing is an excellent paradigm that offers means to deal with such challenges. In this context, replication is a crucial feature for the storage service in the cloud to provide the desperately needed quality of service in terms of availability, performance, and fault tolerance at Big Data scales. The main issue with replication is the *management of consistency* across replicas that may be spread over remote locations. Traditional approaches that ensure *strong consistency* by means of synchronous replication expose the storage clients to wide areas network latency and fail in dealing with Big Velocity challenges of Big Data. In contrast, *eventual consistency* tolerates inconsistency at some points in time but guarantees that all replicas would converge to a consistent state in a future time thus, hiding the network latency from clients. However, this approach may expose clients to far too much stale data.

The work carried out in the context of this Ph.D project addressed the consistency management problematic. We demonstrated that self-adaptivity is necessary to deal with the dynamicity of Big Data applications in the cloud with their *Big Variability* challenges. In this context, we introduced self-adaptive approaches at the storage system level. We showed that these solutions succeeded in enhancing performance, reducing the monetary cost, and saving energy without violating the consistency requirements of the application, whereas static eventual and strong consistency approaches fail. Moreover, and in order to complement our approaches on the system side, we introduced an efficient approach to manage consistency at the Big Data application level. We demonstrated the efficiency of this approach in understanding the consistency requirements of the applications and adapting the consistency

management accordingly. Hereafter, in this chapter, we present our achievements and then highlight the perspectives of our conducted research.

8.1 Achievements

The results achieved in this research can be summarized as the following.

Providing Self–Adaptive Consistency at a High Performance

Over the years, many efforts were dedicated to handle efficiently the tradeoff between consistency and performance within distributed storage systems. However, most of the existing solutions tend to manage consistency in the same manner for all types of applications in a static way. In order to deal with modern Big Data applications running in the cloud and their workloads dynamicity, we introduced a novel approach, *Harmony*, that tunes consistency dynamically and adaptively at runtime according to the application requirements. As to provide efficient tuning, *Harmony* relies on accurate estimation by means of probabilistic computations of stale reads rate in the distributed storage system. The estimation takes into consideration key parameters within the storage system such as the network latency and the observed read and write rates. *Harmony* target therefore, is to keep this stale reads rate below the tolerated stale reads rate of the running application. Moreover, we implemented *Harmony* on top of the Cassandra system as an illustrative system with flexible consistency API.

We conducted two experiment sets on two different platforms. The first set was conducted on Grid'5000 on physical machines and the second set was conducted on Amazon EC2. Results demonstrated that *Harmony* is an efficient approach for cloud workloads. Within both platforms, *Harmony* provides very good performance that is closely similar to the one provided with the eventual consistency level that involves one replica and far better than strong consistency. Moreover, and in contrast to static eventual consistency, *Harmony* does not violate the consistency requirements of the application.

Providing Cost-Efficient Consistency Model

Cloud Computing is an economical–driven paradigm. Therefore, the monetary cost of the storage cost is highly relevant for Big Data Applications. However, most consistency studies tend to focus entirely on performance and availability while neglecting the related financial issues. In this context, we introduced a study that highlighted the tradeoff between consistency and monetary cost in the cloud. We provided a detailed bill of the storage service within the cloud showing the impact of the selected consistency level. In order to exhibit the tight relation between consistency and the monetary cost, we introduced a novel consistency–cost efficiency metric. This metric was leveraged in our cost–efficient approach *Bismar*. *Bismar* adaptively tunes the consistency level at runtime as to achieve substantial cost cuts at a very small fraction of stale reads being read, which is tolerated to a certain degree by numerous applications. In order to achieve its goals, *Bismar* always selects the consistency level with the optimal consistency–cost efficiency value at runtime.

We conducted different sets of experimental evaluations. First, we experimented on Cassandra on Grid'5000 and Amazon EC2 to show the tradeoff between consistency and monetary cost. The results demonstrated our findings about the cost of the storage service that depends on the consistency level. In the second phase, we experimented on *Bismar* deployed on top of Cassandra within two remote sites of Grid'5000. Our results demonstrated how *Bismr* achieved high levels of efficiency at cost reductions that reach up to 31% and performance improvements compared to quorum–based eventual consistency while tolerating a minimal fraction of 3.5% of stale reads that are trivial for many applications.

Analyzing the Impact of Consistency on Energy Consumption

With the increasing power usage within data centers nowadays, an important number of studies is focusing on approaches to reduce the energy consumption. However, consistency impact on storage systems consumptions has hardly been considered. In our study, we addressed this particular issue. We conducted a set of experiments to show the energy consumption of the same workloads with different consistency levels. Therefore, we demonstrated how energy consumption increases as the level of consistency gets higher showing a tradeoff between consistency and energy saving. Moreover, we analyzed the power and the resource usage of an eventually-consistent system (namely Cassandra). We showed that when consistency is eventual there is a variation in usage between the nodes of the storage cluster. In this context, a balanced distribution of data and tasks is far from optimal for reducing the consumption. As a result, we introduced and adaptive configuration of the storage cluster according to the applied consistency. We demonstrated that unbalanced configurations can lead to important energy savings and better consistency with weak consistency levels. In contrast, balanced configurations are the best-fit for strong levels of consistency.

Providing a Customized Consistency Specific to the Application

Consistency management is dealt with almost exclusively at the storage system level within existing solutions. Therefore, all applications are considered the same, and can be differentiated only by their impact on the storage system state. However, applications have different requirements and exhibit distinct consistency needs. In contrast to the existing work, we addressed the consistency management issue at the application level in order to support the management on the system side. First, we introduced a modeling approach of the application behavior when accessing data. We showed that understanding such a behavior is critical for providing a customized consistency. The modeling approach was built relying on machine learning techniques that automatically detect and classify the different states of the application. In addition, we proposed an association mechanism that associates with each state, the most pertinent consistency policy based on the high-level semantics of the application. In a second step, we leveraged the application built model in order to provide a customized consistency approach, named *Chameleon*, for the application at runtime. We introduced an algorithm that observes the current behavior exhibited by the application and predicts its behavior for the next time period. Accordingly, the adequate consistency policy is selected.

Experimental evaluations of the modeling approach demonstrated its efficiency. Our experiments were conducted using traces from Wikipedia as a use case. We showed that

our model achieves more than 96% of accurate recognition of the application behavior without suffering from overfitting problems. In addition, we conducted series of experiments to evaluate *Chameleon* at runtime. We used Cassandra as an underlying storage system deployed on two remote sites in Grid'5000. We demonstrated that *Chameleon* achieves its goal by adapting to the need and the behavior of the application in every time period. Accordingly, it provides the required features in the form of stronger consistency, high performance, reduced cost, and reduced consumption by selecting the most adequate consistency.

8.2 Perspectives

In this Ph.D research, we addressed multiple issues related to consistency and proposed approaches that lead to efficient consistency management for Big Data. In this section, we explore the perspectives and the open doors of our conducted research. Therefore, we show how to exploit the achieved results to work towards even better and more efficient Big Data management in the cloud.

Automated Storage Provisioning in the Cloud with Performance, Consistency, and Cost SLA

The storage service in the cloud is one of the key features. This is mainly because it provides large–scale storage capacities on-demand at a low cost and a good performance to clients. Moreover, it supports full elasticity. Clients can add and release storage resources online according to their needs. However, this elastic provisioning remains non-automated within major cloud vendors.

As a future work, we aim at designing and developing an automated provisioning of storage resources in the cloud according to a pre-established SLA (Service Level Agreement) contract. In contrast to the existing SLAs, this SLA must include the required rate of fresh reads (*ie.* 100 − *the tolerated stale read rate* parameter specified in Chapter 4). In addition, the SLA includes latency requirements and targets the minimization of the monetary cost. In this context, the proposed mechanism will leverage the estimation of stale reads rate introduced in Chapter 4 and the cost computation introduced in Chapter 5 in order to achieve the SLA goals. Accordingly, storage nodes are added or removed automatically and transparently to/from the storage cluster in the cloud.

Energy-Efficient Consistency by means of Self-Adaptive Storage Reconfiguration

As shown in Chapter 6, adapting the storage cluster configuration according to the applied consistency level could lead to important energy savings. As a future work, we plan to leverage the analysis study in Chapter 6 in order to design a self-adaptive reconfiguration of the storage system cluster. The main goal of this approach is to minimize the energy consumption for the runtime workload by automatically resizing the warm and the cold pools of nodes. The proposed approach must monitor the data access to keep track of peak load times as well as compute the read/write ratio. In addition, it must monitor the used consistency levels in the workload. Consequently, all these data are processed in order to determine the best configuration that achieves the adequate performance while reducing

the energy consumption. Thereafter, it dynamically moves nodes from the warm pool to the cold pool and vice versa according to the needs.

QoS Consistency for NoSQL Storage Systems

Current eventually–consistent systems do not provide any guarantees on when all replicas in the storage system would converge to a consistent state. This could be a real problem for many applications as there is no provided certainty about read data. Moreover, if the read data is stale, the question is how stale it could be. In this future work, we plan to design and build a prototype of an eventually–consistent system that provides guarantees on the freshness of data read and ensures that data is consistent after a set of defined deadlines. Furthermore, the proposed system will introduce different levels of guarantees considering the network performance and topology in addition to data location. The storage cluster should consist of a set of zones where a zone is an entity that encapsulates a set of nodes. In practice, a zone can correspond to a geographical location, a datacenter, a rack etc. Moreover, and in order to support multi-tenancy, every application should start its own session when communicating with the storage system. Much like many eventually–consistent systems, the choice of the consistency level is fully flexible, but with support of QoS (Quality of Service). The administrator must specify at the start of every session a set of deadlines for the propagation of data where these deadlines are different depending on which zones are communicating. The data propagation mechanism will be implemented accordingly.

Big Data Management Framework based on the Application Behavior

Big Data Framework. Understanding the behavior of the application is crucial in our context of determining its consistency requirements (as presented in Chapter 7). However, in a broader context, understanding the behavior of the application can lead to a more efficient data and storage management. As a future work, we plan to leverage our approach for modeling the behavior of the application, in order to build an efficient data management framework. This framework will require the application traces as input. The traces must include the data access metrics discussed in Chapter 7 as well as information related to the application clients and the times of failure occurrences in the storage system. These data are further processed by our approach in order to build the application model. Based on the model, the Big Data framework adapts automatically as to provide the most efficient management. Accordingly, policies such as the best replication strategy, the most pertinent consistency policy, the best strategy to prevent failures or the best strategy to tolerate failures (considering consistency meanwhile), and the best provisioning approach of the storage resources are determined. As a result, the Big Data framework adapts specifically to the application needs providing efficiency at large scale with full automation, which is becoming critical for Big Data applications.

Hadoop-based Application Modeling. Leveraging the application behavior modeling to provide Big Data framework implies the need for processing large traces that contain larger volumes of data than presented in Chapter 7. Moreover, and for modeling efficiency and accuracy, the traces should contain data covering many years. As a result, processing these large datasets using single-threaded machine learning algorithms becomes inefficient. In this context, we plan to build a Hadoop based model. Hereafter, traces processing will be implemented using

the MapReduce programming model. Moreover, the machine learning algorithms must be implemented using the same programming model. Subsequently, we plan to investigate using *Mahout* [22], the scalable machine learning and data mining library for Hadoop.

Bibliography

[1] *3D data management: Controlling data volume, variety and velocity*. 2013. URL: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[2] *5 million terabytes of data*. 2013. URL: http://myhumannetwork.com/5-million-terabytes-of-data/.

[3] Daniel J. Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story". In: *Computer* 45 (2012), pp. 37–42.

[4] *About Data Consistency in Cassandra*. 2013. URL: http://www.datastax.com/docs/1.0/dml/data_consistency.

[5] Sarita V. Adve and Mark D. Hill. "Weak ordering - a new definition". In: *SIGARCH Comput. Archit. News* 18.3a (1990), pp. 2–14.

[6] Mustaque Ahamad et al. "Causal Memory: Definitions, Implementation, and Programming". In: *Distributed Computing* 9.1 (1995), pp. 37–49.

[7] *Amazon DynamoDB*. 2013. URL: http://aws.amazon.com/dynamodb/.

[8] *Amazon Elastic Block Store (Amazon EBS)*. 2013. URL: http://aws.amazon.com/ebs/.

[9] *Amazon Elastic Compute Cloud (Amazon EC2)*. 2013. URL: http://aws.amazon.com/ec2/.

[10] *Amazon Elastic MapReduce (Amazon EMR)*. 2013. URL: http://aws.amazon.com/elasticmapreduce/.

[11] *Amazon Glacier*. 2013. URL: http://aws.amazon.com/glacier/.

[12] *Amazon Simple Queue Service (Amazon SQS)*. 2013. URL: http://aws.amazon.com/sqs/.

[13] *Amazon Simple Storage Service (Amazon S3)*. 2013. URL: http://aws.amazon.com/s3/.

[14] *Amazon Web Services (Amazon AWS)*. 2013. URL: http://aws.amazon.com/.

[15] *Amazon.com*. 2013. URL: http://www.amazon.com/.

[16] Hrishikesh Amur et al. "Robust and flexible power-proportional storage". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 217–228.

[17] Eric Anderson et al. “What consistency does your key-value store actually provide?” In: *Proceedings of the Sixth international conference on Hot topics in system dependability*. HotDep’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–16.

[18] *Apache Cassandra*. 2013. URL: http://cassandra.apache.org/.

[19] *Apache CouchDB*. 2013. URL: http://couchdb.apache.org/.

[20] *Apache Hadoop*. 2013. URL: http://hadoop.apache.org/.

[21] *Apache HBase*. 2013. URL: http://hbase.apache.org/.

[22] *Apache Mahout: scalable machine learning and data mining*. 2013. URL: http://mahout.apache.org/.

[23] *Apache Thrift*. 2013. URL: http://thrift.apache.org/.

[24] *APC Corporation*. 2013. URL: http://www.apc.com.

[25] Michael Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. EECS Department, University of California, Berkeley, 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[26] Jason Baker et al. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234.

[27] *Benchmarking Cassandra Scalability on AWS - Over a million writes per second*. 2013. URL: http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.

[28] Fabrício Benevenuto et al. “Characterizing user behavior in online social networks”. In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. IMC ’09. Chicago, Illinois, USA: ACM, 2009, pp. 49–62.

[29] David Bermbach and Stefan Tai. “Eventual consistency: How soon is eventual? An evaluation of Amazon S3’s consistency behavior”. In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. MW4SOC ’11. Lisbon, Portugal: ACM, 2011.

[30] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[31] *Big Data – What Is It?* 2013. URL: http://www.sas.com/big-data/.

[32] Roberto Bisiani, Andreas Nowatzyk, and Mosur Ravishankar. “Coherent Shared Memory on a Distributed Memory Machine”. In: *ICPP (1)*. 1989, pp. 133–141.

[33] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A training algorithm for optimal margin classifiers”. In: *Proceedings of the fifth annual workshop on Computational learning theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pp. 144–152.

[34] E. Brewer. “CAP twelve years later: How the "rules" have changed”. In: *Computer* 45.2 (2012), pp. 23 –29.

[35] Eric A. Brewer. “Towards robust distributed systems (abstract)”. In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. PODC ’00. Portland, Oregon, United States: ACM, 2000.

[36] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Gener. Comput. Syst.* 25.6 (2009), pp. 599–616.

[37] Brad Calder et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 143–157.

[38] Philip H. Carns et al. "PVFS: A Parallel File System for Linux Clusters". In: *IN PROCEEDINGS OF THE 4TH ANNUAL LINUX SHOWCASE AND CONFERENCE*. MIT Press, 2000, pp. 391–430.

[39] J. D. Case et al. *Simple Network Management Protocol (SNMP)*. United States, 1990.

[40] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *Proceedings of the 7th conference on usenix symposium on operating systems design and implementation*. 2006, pp. 205–218.

[41] Xue wen Chen and Jong Cheol Jeong. "Enhanced recursive feature elimination". In: *Sixth International Conference on Machine Learning and Applications, 2007. ICMLA 2007.* 2007, pp. 429–435.

[42] Navraj Chohan et al. "See spot run: using spot instances for mapreduce workflows". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. Boston, MA, 2010.

[43] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154.

[44] Brian F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proc. VLDB Endow.* 1 (2008), pp. 1277–1288.

[45] James C. Corbett et al. "Spanner: Google's globally-distributed database". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264.

[46] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297.

[47] *Cost of Power in Large-Scale Data Centers*. 2013. URL: `http://perspectives.mvdirona.com`.

[48] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[49] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. Stevenson, Washington, USA, 2007, pp. 205–220.

[50] Ewa Deelman et al. "The cost of doing science on the cloud: the Montage example". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, pp. 51–62.

[51] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm". In: *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B* 39.1 (1977), pp. 1–38.

[52] *Dstat: Versatile resource statistics tool*. 2013. URL: http://linux.die.net/man/1/dstat.

[53] Michel Dubois, Christoph Scheurich, and Faye Briggs. "Memory access buffering in multiprocessors". In: *25 years of the international symposia on Computer architecture (selected papers)*. ISCA '98. Barcelona, Spain: ACM, 1998, pp. 320–328.

[54] *EATON Corporation*. 2013. URL: http://www.eaton.com/.

[55] EPA. *EPA Report to Congress on Server and Data Center Energy Efficiency*. Tech. rep. U.S. Environmental Protection Agency, 2007.

[56] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 226–231.

[57] *Eucalyptus*. 2013. URL: http://www.eucalyptus.com/.

[58] *Facebook: Power use in Prineville data center more than doubled last year*. 2013. URL: http://www.oregonlive.com/silicon-forest/index.ssf/2013/07/facebook_power_use_in_prinevil.html.

[59] *Facebook Statistcs*. 2013. URL: http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

[60] Ian Foster. "What is the Grid? - a three point checklist". In: *GRIDtoday* 1.6 (July 2002).

[61] Ian Foster and Carl Kesselman. "The grid". In: ed. by Ian Foster and Carl Kesselman. Morgan Kaufmann Publishers Inc., 1999. Chap. Computational grids, pp. 15–51.

[62] Simson L. Garfinkel. *An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS*. Tech. rep. Technical Report, Harvard University, 2007.

[63] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *SIGOPS - Operating Systems Review* 37.5 (2003), pp. 29–43.

[64] David K. Gifford. "Weighted voting for replicated data". In: *Proceedings of the seventh ACM symposium on Operating systems principles*. SOSP '79. Pacific Grove, California, United States: ACM, 1979, pp. 150–162.

[65] S. Gilbert and N. Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (2012), pp. 30 –36.

[66] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services". In: *ACM SIGACT News*. 2002.

[67] *Google App Engine*. 2012. URL: http://code.google.com/appengine/.

[68] Isabelle Guyon et al. "Gene Selection for Cancer Classification using Support Vector Machines". In: *Mach. Learn.* 46.1-3 (2002), pp. 389–422.

[69] *Hadoop Distributed File System (HDFS)*. 2013. URL: http://hadoop.apache.org/.

[70] *Hadoop running in production on the Yahoo! Search Webmap*. 2013. URL: http://developer.yahoo.com/blogs/ydn/hadoop-running-production-yahoo-search-webmap-7307.html.

[71] *HDD vs SSD*. 2013. URL: http://www.diffen.com/difference/HDD_vs_SSD.

[72] Maurice Herlihy. "A quorum-consensus replication method for abstract data types". In: *ACM Trans. Comput. Syst.* 4.1 (1986), pp. 32–53.

[73] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492.

[74] *HP Cloud*. 2013. URL: `https://www.hpcloud.com/`.

[75] Patrick Hunt et al. "ZooKeeper: wait-free coordination for internet-scale systems". In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010.

[76] Shadi Ibrahim, Bingsheng He, and Hai Jin. "Towards Pay-As-You-Consume Cloud Computing". In: *Proceedings of the 2011 IEEE International Conference on Services Computing (SCC'11)*. SCC '11. Washington, DC, USA, 2011, pp. 370–377.

[77] Shadi Ibrahim et al. "Adaptive Disk I/O Scheduling for MapReduce in Virtualized Environment". In: *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. ICPP '11. Taipei, Taiwan, 2011, pp. 335–344.

[78] *Java Machine Learning Library (Java-ML)*. 2013. URL: `http://java-ml.sourceforge.net/`.

[79] Y. Jégou, S. Lantéri, J. Leduc, et al. "Grid'5000: a large scale and highly reconfigurable experimental Grid testbed." In: *Intl. Journal of High Performance Comp. Applications* 20.4 (2006), pp. 481–494.

[80] David Karger et al. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663.

[81] Rini T. Kaushik and Milind Bhandarkar. "GreenHDFS: towards an energy-conserving, storage-efficient, hybrid Hadoop compute cluster". In: *Proceedings of the 2010 international conference on Power aware computing and systems*. HotPower'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–9.

[82] Tim Kraska et al. "Consistency rationing in the cloud: pay only when it matters". In: *Proc. VLDB Endow.* 2 (2009), pp. 253–264.

[83] Rivka Ladin, Barbara Liskov, and Liuba Shrira. "Lazy replication: exploiting the semantics of distributed services". In: *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*. PODC '90. Quebec City, Quebec, Canada: ACM, 1990, pp. 43–57.

[84] Rivka Ladin et al. "Providing high availability using lazy replication". In: *ACM Trans. Comput. Syst.* 10 (Nov. 1992), pp. 360–391.

[85] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *SIGOPS Oper. Syst. Rev.* 44 (2010), pp. 35–40.

[86] Willis Lang and Jignesh M. Patel. "Energy management for MapReduce clusters". In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 129–139.

[87] Cheng Li et al. "Making geo-replicated systems fast as possible, consistent when necessary". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278.

[88] *LIBSVM – A Library for Support Vector Machines*. 2013. URL: http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[89] Huan Liu. "Cutting MapReduce Cost with Spot Market". In: *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*. HotCloud'11. Portland, OR, 2011.

[90] Rui Liu, Ashraf Aboulnaga, and Kenneth Salem. "DAX: a widely distributed multitenant storage service for DBMS hosting". In: *Proceedings of the 39th international conference on Very Large Data Bases*. PVLDB'13. Trento, Italy: VLDB Endowment, 2013, pp. 253–264.

[91] Wyatt Lloyd et al. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 401–416.

[92] Wyatt Lloyd et al. "Stronger semantics for low-latency geo-replicated storage". In: *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 313–328.

[93] J. B. MacQueen. "Some Methods for Classification and Analysis of MultiVariate Observations". In: *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. Ed. by L. M. Le Cam and J. Neyman. Vol. 1. University of California Press, 1967, pp. 281–297.

[94] Warren S. McCulloch and Walter Pitts. "Neurocomputing: foundations of research". In: ed. by James A. Anderson and Edward Rosenfeld. MIT Press, 1988. Chap. A logical calculus of the ideas immanent in nervous activity, pp. 15–27.

[95] *Microsoft Windows Azure Cloud Services*. 2013. URL: http://www.windowsazure.com/.

[96] *mongoDB*. 2013. URL: http://www.mongodb.org/.

[97] Jesús Montes et al. "Finding order in chaos: a behavior model of the whole grid". In: *Concurrency and Computation: Practice and Experience* 22.11 (2010), pp. 1386–1415.

[98] *MPICH: High-Performance Portable MPI*. 2013. URL: http://www.mpich.org/.

[99] *Nimbus*. 2013. URL: http://www.nimbusproject.org/.

[100] *NodeTool*. 2012. URL: http://wiki.apache.org/cassandra/NodeTool.

[101] *OCCI: Open Cloud Computing Interface*. 2013. URL: http://occi-wg.org/.

[102] *OpenNebula: Open Source Data Center Virtualization*. 2013. URL: http://opennebula.org/.

[103] *OpenStack*. 2013. URL: http://www.openstack.org/.

[104] Mayur R. Palankar et al. "Amazon S3 for science grids: a viable solution?" In: *Proceedings of the 2008 international workshop on Data-aware distributed computing*. DADC '08. Boston, MA, USA: ACM, 2008, pp. 55–64.

[105] David A. Patterson, Garth Gibson, and Randy H. Katz. "A case for redundant arrays of inexpensive disks (RAID)". In: *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. SIGMOD '88. Chicago, Illinois, USA: ACM, 1988, pp. 109–116.

[106] R. Peglar. *Eliminating planned downtime: the real impact and how to avoid it*. 2012. URL: http://findarticles.com/p/articles/mi_m0BRZ/is_5_24/ai_n6095515/.

[107] *Redis*. 2013. URL: http://redis.io/.

[108] *Revolutionary Methods to Handle Data Durability Challenges for Big Data*. 2013. URL: http://www.intel.com/content/www/us/en/big-data/big-data-amplidata-storage-paper.html.

[109] *Riak*. 2013. URL: http://basho.com/riak/.

[110] Yasushi Saito and Marc Shapiro. "Optimistic replication". In: *ACM Comput. Surv.* 37.1 (2005), pp. 42–81.

[111] Sherif Sakr et al. "CloudDB AutoAdmin: Towards a Truly Elastic Cloud-Based Data Store". In: *Proceedings of the 2011 IEEE International Conference on Web Services*. ICWS '11. IEEE Computer Society, 2011, pp. 732–733.

[112] Christoph Scheurich and Michel Dubois. "Concurrent Miss Resolution in Multiprocessor Caches". In: *ICPP (1)*. 1988, pp. 118–125.

[113] Frank Schmuck and Roger Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". In: *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*. 2002, pp. 231–244.

[114] Philip Schwan. "Lustre: Building a File System for 1,000-node Clusters". In: *PROCEEDINGS OF THE LINUX SYMPOSIUM*. 2003, p. 9.

[115] Marc Shapiro and Bettina Kemme. "Eventual Consistency". Anglais. In: *Encyclopedia of Database Systems (online and print)*. Ed. by M. Tamer Ózsu and Ling Liu. springer, 2009.

[116] Ion Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160.

[117] Diary R. Suleiman, Muhammed A. Ibrahim, and Ibrahim I. Hamarash. *DYNAMIC VOLTAGE FREQUENCY SCALING (DVFS) FOR MICROPROCESSORS POWER AND ENERGY REDUCTION*.

[118] Ann T. Tai and John F. Meyer. "Performability Management in Distributed Database Systems: An Adaptive Concurrency Control Protocol". In: *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. MASCOTS '96. IEEE Computer Society, 1996.

[119] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[120] *The Go Programming Language*. 2013. URL: http://golang.org/.

[121] Robert H. Thomas. "A Majority consensus approach to concurrency control for multiple copy databases". In: *ACM Trans. Database Syst.* 4.2 (1979), pp. 180–209.

[122] *Under the Hood: Scheduling MapReduce jobs more efficiently with Corona*. 2013. URL: https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920.

[123] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. "Wikipedia Workload Analysis for Decentralized Hosting". In: *Elsevier Computer Networks* 53.11 (2009), pp. 1830–1845. URL: http://www.globule.org/publi/WWADH_comnet2009.html.

[124] Werner Vogels. "Eventually consistent". In: *Commun. ACM* (2009), pp. 40–44.

[125] *Voldemort*. 2013. URL: http://www.project-voldemort.com/voldemort/.

[126] Hiroshi Wada et al. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective". In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 2011, pp. 134–143.

[127] Hongyi Wang et al. "Distributed systems meet economics: pricing in the cloud". In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*. HotCloud'10. Boston, MA: USENIX Association, 2010.

[128] Ximei Wang et al. "An Application-Based Adaptive Replica Consistency for Cloud Storage". In: *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. 2010, pp. 13 –17.

[129] Sage Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*. 2006.

[130] Aaron Weiss. "Computing in the clouds". In: *netWorker* 11.4 (2007), pp. 16–25.

[131] *Weka 3: Data Mining Software in Java*. 2013. URL: http://www.cs.waikato.ac.nz/ml/weka/.

[132] Jason Weston et al. "Use of the zero-norm with linear models and kernel methods". In: *Journal of Machine Learning Research* 3 (2003), pp. 1439–1461.

[133] *What Does 'Big Data' Mean?* 2013. URL: http://cacm.acm.org/blogs/blog-cacm/155468-what-does-big-data-mean/fulltext.

[134] *What is Cloud Computing?* 2013. URL: http://aws.amazon.com/what-is-cloud-computing/.

[135] *Wikipedia*. 2013. URL: http://en.wikipedia.org/.

[136] *World's data will grow by 50X in next decade, IDC study predicts*. 2013. URL: http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts.

[137] *Yahoo Cloud Serving Benchmark*. 2013. URL: https://github.com/brianfrankcooper/YCSB/wiki.